Optimización de Funciones de DSP para Procesador con Instrucciones SIMD

Proyecto de Sistemas Informáticos
Curso 2007/2008

Facultad de Informática
Univeridad Complutense de Madrid

Autor
Jon García de Salazar Ochoa

Profesor Director
Luis Piñuel Moreno

Resumen

CoolFlux es un DSP de ultra bajo consumo desarrollado por NXP que se utiliza en reproductores de audio portátiles, audífonos o para el procesamiento con fines biomédicos. Para los usos anteriormente mencionados, el consumo tiene una gran importancia, por eso, en una nueva revisión de este DSP, denominada *CoolFlux Complex*, se han incorporado instrucciones SIMD. Este nuevo tipo de instrucciones permiten aplicar una operación sobre dos datos al mismo tiempo, para hacerlo, *CoolFlux Complex*, es capaz almacenar en una palabra de memoria dos datos.

En este proyecto se han optimizado algunas de las funciones típicas de los DSP mediante la utilización de instrucciones SIMD. Las funciones optimizadas incluyen filtros FIR, filtros Biquad, la transformada inversa discreta del coseno, así como varias operaciones aritméticas básicas con vectores. El objetivo del proyecto es el de evaluar el impacto en el rendimiento al utilizar el nuevo conjunto de instrucciones. Un aumento del rendimiento del DSP implicaría un menor consumo, al necesitar menos tiempo para procesar la misma cantidad de datos. Esta optimización se ha llevado a cabo sobre la mathlib de *CoolFlux Complex*, escrita en C. Ninguna de las funciones que contiene la mathlib utiliza instrucciones SIMD así que las funciones de partida están escritas para un tipo de datos llamado *fix* (datos en punto fijo). En el caso de la transformada inversa discreta del coseno, no existía ninguna función escrita para *CoolFlux Complex* así que hubo que escribir una función para el tipo de datos *fix* para poder hacer comparaciones de rendimiento con la versión que utiliza instrucciones SIMD.

Los resultados obtenidos tras la optimización de las funciones son muy diferentes dependiendo del algoritmo que implementa cada función. En los casos en los que los algoritmos no requieren muchos movimientos de datos, los resultados son buenos, en cambio, si los algoritmos necesitaban mucho movimiento de datos, los resultados eran muy malos. Donde realmente destaca el nuevo conjunto de instrucciones es en el procesamiento de dos canales, en los que unos datos de un canal son independientes del otro, en estos casos la ganancia es muy significativa. El motivo de estos resultados es que hay muy pocas instrucciones que permitan organizar los datos SIMD y las pocas que hay, son muy generales y su uso requiere muchos ciclos del DSP.

Abstract

CoolFlux is an ultra low power consumption DSP developed by NXP used in portable audio players, hearing devices or biomedical processing. For that kind of uses, consumption is very important, for that reason in a new revision of this DSP, called *CoolFlux Complex*, SIMD instructions have been added. This new set of instructions allow to operate with two values at the same time, to do so, *CoolFlux Complex* is capable of storing two values into one single memory word.

In this project some typical DSP functions has been optimized by using SIMD instructions. The optimized functions include FIR filters, filters biquad, inverse discrete cosine transform, as well as some basic vector operations. The goal of this project is to evaluate the impact on performance when using the new set of instructions. An increase in the DSP performance implies minor power consumption as require less time to process the same amount of data. This optimization has been done on the *CoolFlux Complex* mathlib, written in C. None of the functions contained in the mathlib uses SIMD instructions, so optimized functions are based on the old functions written using *fix* data type (fixed point data). In the case of inverse discrete cosine transform, there was not any function written for *CoolFlux Complex*, so a function that uses fix data type was needed and writen in order to make performance comparisons beetwen the version that uses fix data type and the one that uses SIMD instructions.

The results after the optimization of the functions are very different depending on the algorithm that implements each function. In cases that the algorithm does not require many data movements, results are good, however, if the algorithm need many data movements, results were very poor. Where the new set of instruction stands out is in the processing of two channels, where data from one channel is independent of the other one, in these cases the speedup is very significant. The reason for these results is that there are few instructions to organize SIMD data and those that exist are very general and their use requires many DSP cycles.

Lista de palabras

DSP, SIMD, CoolFlux Complex, optimización, operaciones con vectores, IDCT, filtros

Jon García de Salazar Ochoa autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, la memoria desarrollada en este proyecto.

Madrid, a 1 de Julio de 2008

Fdo. Jon García de Salazar Ochoa

Agradecimientos

Quiero dar las gracias a Luis Piñuel director del proyecto, por haber confiado en mí para la realización de este proyecto, a Franciso Barat y Jeroen Coninx, por su inestimable ayuda durante mi estancia en NXP, y a mis padres, sin su apoyo esto habría sido mucho más difícil. Muchas gracias a todos.

Índice General

Índice de Figuras

Índice de Cuadros

1. Introducción

El Procesamiento Digital de Señales es un área de la ingeniería que se dedica al análisis y procesamiento de señales (audio, voz, imágenes, video) discretas, aunque comúnmente las señales en la naturaleza nos llegan en forma analógica. Si casi todo en la naturaleza se mueve, se basa y se desarrolla de forma analógica, ¿qué se pretende transformando lo analógico en digital? Este procesamiento se realiza sobre señales digitales por diferentes razones:

- Una señal digital es más fácil de procesar que una analógica.
- Las señales son digitalizadas para facilitar su transmisión o almacenamiento.
- Es posible realizar mediante procesamiento digital acciones imposibles de obtener mediante el procesamiento analógico (por ejemplo, filtros con respuesta de frecuencia arbitraria).

El procesamiento se hace en forma digital porque es más cómodo de realizar y más barato de implementar. Además las señales digitales requieren usualmente menos ancho de banda y pueden ser comprimidas. Sin embargo, hay pérdida inherente de información al convertir la información continua en discreta.

Cualquier procesador de propósito general es capaz de realizar este tipo de procesos, pero para poder hacerlos de una manera eficiente es necesario emplear procesadores especializados en el Procesamiento Digital de Señales. Estos procesadores se llaman *Procesadores Digitales de Señales* (*DSP*). Un DSP es un sistema basado en un procesador o microprocesador que posee un juego de instrucciones, un hardware y un software optimizados para aplicaciones que requieran operaciones numéricas a muy alta velocidad. Debido a esto es especialmente útil para el procesado y representación de señales analógicas en tiempo real: en un sistema que trabaje de esta forma se reciben muestras, normalmente provenientes de un conversor analógico-digital. Se ha dicho que puede trabajar con señales analógicas, pero es un sistema digital, por lo tanto necesitará un conversor analógico/digital a su entrada y digital/analógico en la salida. Como todo sistema basado en procesador programable necesita una memoria donde almacenar los datos con los que trabajará y el programa que ejecuta. Si se tiene en cuenta que un DSP puede trabajar con varios datos en paralelo y un diseño e instrucciones específicas

para el procesado digital, se puede dar una idea de su enorme potencia para este tipo de aplicaciones. Estas características constituyen la principal diferencia de un DSP y otros tipos de procesadores.

En éste proyecto se ha trabajado con un procesador en concreto, una evolución de *CoolFlux* llamado *CoolFlux Complex*, un DSP de alta precisión y ultra bajo consumo, desarrollado por *NXP*. *CoolFlux* ha sido diseñado junto con un compilador C muy eficiente en la optimización del paralelismo a nivel de instrucción (ILP). El compilador puede explotar todo el paralelismo del núcleo y genera un código muy eficiente tanto en número de ciclos como en tamaño. *CoolFlux Complex*, introduce una serie de modificaciones que sirven para acelerar la ejecución de determinadas operaciones. Algunas de las modificaciones introducidas en el *CoolFlux Complex* son la aparición de dos nuevos tipos de datos. Estos son dos tipos empaquetados de datos: *Simd* y *Complex*. Los datos *Complex* almacenan dos datos del tipo *fix*, que son datos expresados en punto fijo, de la mitad de precisión. Uno de los datos corresponde a la parte real de un número y el otro a la imaginaria. La ventaja que tiene el uso de este tipo de datos es que la multiplicación de números complejos está implementada como una instrucción hardware, lo que permite realizar ésta operación seis veces más rápido. Los datos *Simd*, acrónimo de *"Single Instruction Multiple Data"*, también almacenan dos datos del tipo *fix* de la mitad de precisión, es decir, en una sola palabra de memoria se almacenan dos datos, uno de ellos se sitúa en la parte más significativa y el otro en la parte menos significativa. Las unidades funcionales de *CoolFlux* han sido modificadas para que puedan aplicar la misma operación de forma simultánea a ambos datos. Estas modificaciones de las unidades funcionales traen consigo la incorporación de un conjunto de instrucciones especifico para el manejo de este tipo de datos. Que los tipos empaquetados contengan datos que son de la mitad de precisión, puede ser un inconveniente en determinadas aplicaciones, en las que la exactitud en los resultados tenga una gran importancia. Sin embargo, en las aplicaciones en las que el tiempo es crítico, éstas mejoras aceleran de una manera significativa muchos procesos. Además, el aumento en el rendimiento supone un ahorro en el consumo, algo que supone un gran beneficio, ya que *Coolflux Complex* es utilizado en áreas donde el consumo es un factor muy importante.

El objetivo de éste proyecto es, utilizando el nuevo tipo de datos *Simd*, optimizar el código de algunas de las funciones de la librería matemática, escrita en C, del *CoolFlux* para su uso en el *CooFlux Complex*. Y evaluar el impacto que el tipo *Simd* tiene sobre el rendimiento de esas funciones. Al ser la primera vez que *CoolFlux*

incorpora este tipo de funciones, no se tenía una idea clara sobre los beneficios que las modificaciones introducidas en esta versión iban a ofrecer. Idealmente, el rendimiento con este tipo de datos es del doble, ya que podemos aplicar una operación sobre dos valores al mismo tiempo.La optimización de las funciones, toman como funciones de partida las existentes en la *mathlib* que están escritas para datos *fix,* los cuales representan números en punto fijo, ya que en las librerías de *CoolFlux Complex* no disponía de funciones escritas utilizando datos SIMD. En dicha librería matemática se han optimizado, más concretamente, funciones como un filtro *FIR* en varias versiones, un filtro *Biquad*, Operaciones con vectores y la transformada inversa del coseno (*iDCT*), de las que se hablará con más detalle más adelante en el capítulo 4.

2. CoolFlux Complex

CoolFlux Complex es un DSP de propósito general de alta precisión, ultra bajo consumo y de bajo coste con una arquitectura dual Harvard y Dual MAC que tiene soporte para aritmética fraccional así como un soporte de altamente eficiente para los tipos de datos y operaciones de ANSI C.

CoolFlux es un DSP para aplicaciones de ultra bajo consumo. El núcleo fue desarrollado para aplicaciones de audio portátil, incluyendo la codificación y decodificación de audio, algoritmos de mejora de audio y de eliminación de ruidos.

Éstas son algunas de las aplicaciones de *CoolFlux Complex*:

- Reproductores de audio portátil
- Audífonos
- Mejora de la voz y cancelación de ruidos.
- Procesamiento de sensores y procesamiento biomédico
- Uso de varios núcleos para un rendimiento mayor (Sonido de alta calidad)

2.1. Arquitectura

CoolFlux Complex es un DSP capaz de realizar dos MACs, dos operaciones de memoria y dos actualizaciones de punteros por ciclo, esto hace de *CoolFlux Complex* un DSP muy eficiente en ciclos para aplicaciones de computación intensiva. Muchas de las operaciones aritméticas pueden realizarse de forma escalar o con datos empaquetados. Los datos empaquetados pueden ser Complex o Simd. La distinción entre ambos modos se realiza mediante un bit en el registro de estado. De acuerdo con lo explicado, *CoolFlux Complex* es capaz de hacer cuatro multiplicaciones de 12-bits en paralelo.

CoolFlux Complex es una arquitectura de load/store, esto implica que:

- Todos los operandos deben ser movidos a registros de forma explícita.
- Todas las unidades de ejecución cogen sus valores de entradas de registros y guardan sus resultados en un registro.
- Todos los resultados producidos deben ser movidos a memoria de forma explicita.

Las siguientes figuras muestran la arquitectura de *CoolFlux Complex*. En ellas puedes encontrar:

- La unidad de control de programa con los registros *pc*, *sr*, *isr*, *lr*, *ilr* y 16M palabras de 32 bits de memoria de programa.
- Las unidades X e Y de direccionamiento, que contienen:
  - Las unidades XAGU e YAGU de generación de direcciones.
  - Los registros xptr, xstep, xmod y sp.
  - Los registros *yptr*, *ystep* e *ymod*.
- Las 16M palabras de 24 bits de la memoria-X, memoria-Y y la memoria-I/O.
- El bloque DMA.
- Los buses Xbus e Ybus de 24 bits.
- Los bancos de registros X e Y con los registros *x0*, *x1*, *y0* e *y1* de 24 bits.
- Los bancos de registros acumuladores A y B con los registros *a0*, *a1*, *b0* y *b1* de 56 bits.
- La ruta de datos con:
  - Las unidades XMUL e YMUL.
  - La unidad pre-add/substract del multiplicador XMUL.
  - Las unidades XALU e YALU.
- Las dos unidades RSS (redondeo, saturación y selección).
- Los bloques de conversión de las diferentes anchuras de datos.

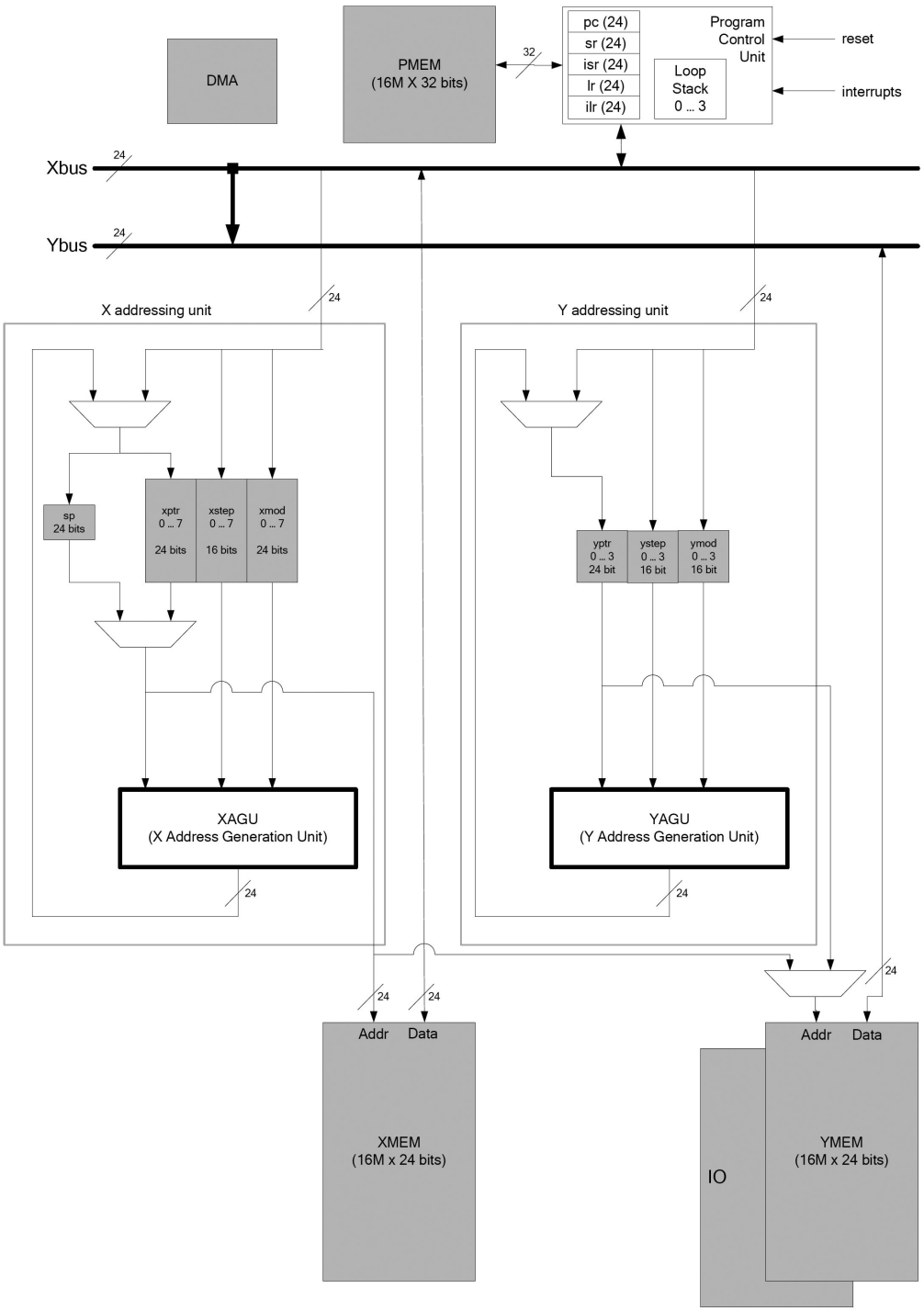

Ilustración 1:Unidades de direccionamiento, buses Xbus e Ybus, memorias, DMA y unidad de control de programa.

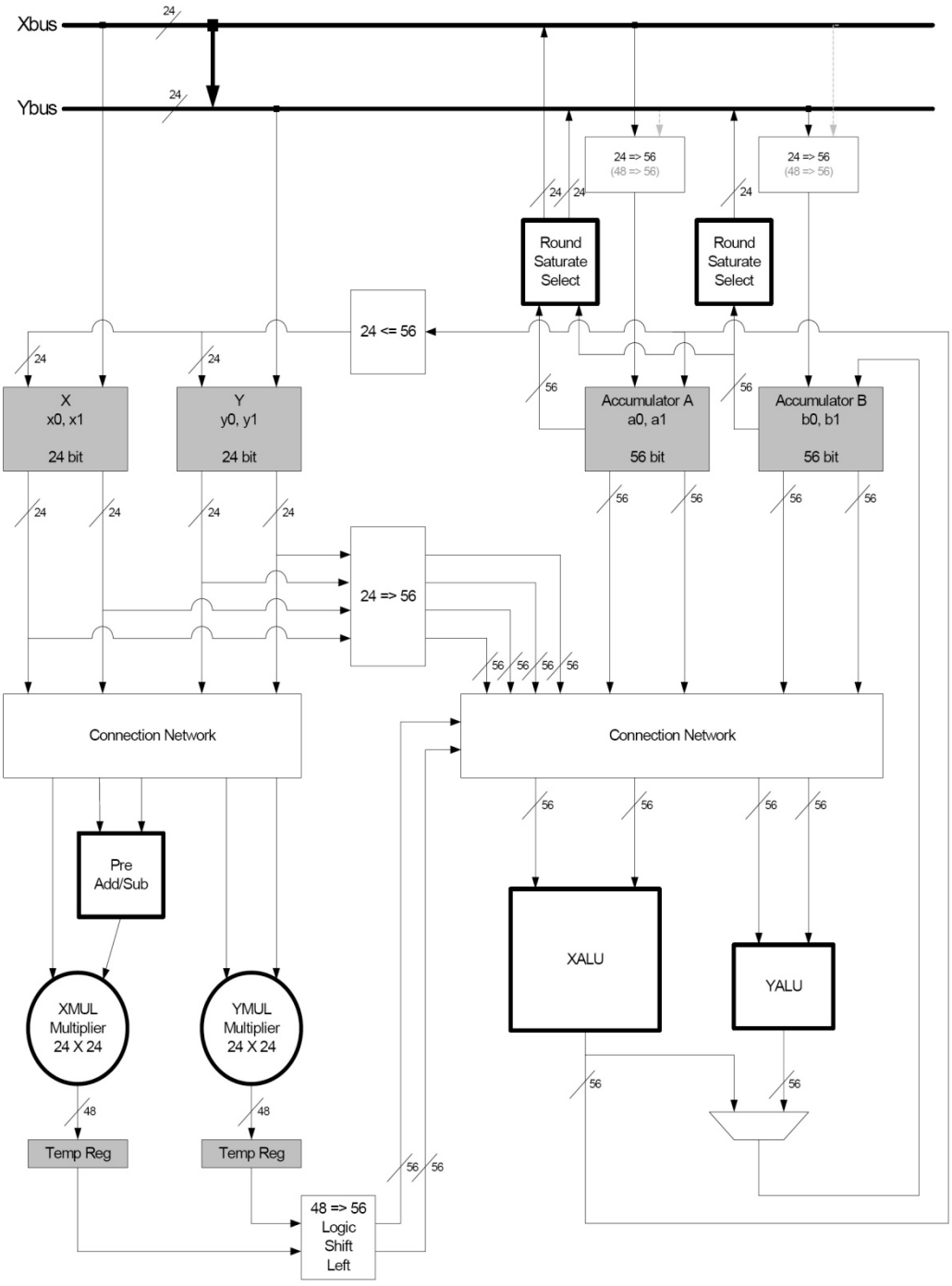

Ilustración 2:Ruta de Datos, unidades RSS y buses Xbus e Ybus.

2.2. Registros

Los registros en el *CoolFlux Complex* tienen una gran importancia al tratarse de una arquitectura de load/store. Todas las unidades de ejecución toman sus entradas de los registros y guardan los resultados en los registros. Los accesos a memoria son considerados como operaciones. Un store toma datos de los registros y los almacena en memoria, mientras que la operación de load lee un valor de la memoria y la almacena en un registro.

Los registros de *CoolFlux Complex* están organizados en bancos de registros. Todos los registros de un mismo banco tienen el mismo nombre, pero un índice diferente. Las operaciones trabajan sobre registros individuales, pero en la descripción de la sintaxis se utilizan los nombres de los bancos de registros. Esto es porque todos los registros de un banco de registros pueden ser usados de la misma manera en una operación. Se deben usar registros individuales como fuente y destino de operaciones reales.

2.3. La Ruta de Datos

La ruta de datos consta de:

- 4 bancos de registros A, B, X e Y conectados a las unidades aritméticas.
- Dos multiplicadores XMUL e YMUL (24x24bits). Que pueden funcionar de varias formas (multiplicación con signo, sin signo, de números complejos y SIMD)
    - La unidad XMUL tiene una unidad de suma/resta previa.
- 2 Unidades ALU XALU e YALU.

La arquitectura de *CoolFlux Complex* muestra una clara simetría. Sin embargo, las llamadas X-operaciones (las que usan la XALU y XMUL) son más elaboradas.

2.3.1. Multiplicadores

CoolFlux Complex posee dos unidades de multiplicación, XMUL e YMUL. Ambas unidades tienen dos entradas de 24 bits. Leen sus operandos de los bancos de registros X e Y. La unidad XMUL puede ser usada para multiplicaciones *signed/signed*, *unsigned/signed*, *unsigned/unsigned*, complejasySIMD.

Cuando se esta haciendo una multiplicación de complejos, se considera que las entradas tienen una parte imaginaria y otra parte real, ambas de 12 bits. Cuando se hace una multiplicación SIMD, se interpretan las entradas como datos compuestos por dos subpartes de 12 bits cada una.

A la unidad XMUL le precede la unidad de pre-suma/resta. Esta unidad puede usarse de tres maneras diferentes: paso, suma y resta. Para una multiplicación simple se usa en el modo paso en el que copia una de sus entradas a la salida, esta salida se multiplica después en la XMUL con otro registro XY. En caso de usarla en modo suma (o resta), toma dos registros XY y los suma (resta) y el resultado lo multiplica con un tercer registro XY. Esta unidad no puede ser utilizada en modo de multiplicación compleja o SIMD.

En todos los casos, el resultado de una multiplicación es un número de 48 bits. Antes de usarse en las ALUs, el número es convertido a 56 bits. Éste valor de 56 bits se almacena en un registro temporal y en el siguiente ciclo del procesador será usado como un operando para la acumulación a través de una de las ALUs (XALU para resultados de XMUL, YALU para resultados de YMUL).

A los multiplicadores se accede siempre con operaciones MAC (multiplicación-acumulación), que requiere de dos ciclos del procesador para finalizar. En el primer ciclo se realiza la multiplicación en XMUL o YMUL. En el segundo ciclo, el resultado se acumula mediante la XALU o la YALU. En cada ciclo del procesador, se puede lanzar una nueva operación MAC, dado que la multiplicación se puede hacer en paralelo a la acumulación de la operación MAC anterior.

2.4. Unidades Aritmético Lógicas

La XALU y la YALU siempre operan sobre datos de 56 bits y generan resultados de 56 bits. Toman sus entradas de los bancos de registros A, B, X e Y o de los registros temporales de las salidas de los multiplicadores. Todas estas entradas son convertidas, si es necesario, a 56 bits. El resultado de la XALU puede almacenarse en los bancos de registros A, B, X e Y; el resultado de la YALU puede ser almacenada en un registro B.

Las operaciones permitidas por la unidad XALU son:

- Suma, resta, (suma y división por 2), (resta y división por 2), suma reducción, resta reducción.
- Y lógica, O lógica, XOR lógica.
- Paso de división
- Min, max
- Negación, exponente, valor absoluto, extensión de signo
- Desplazamiento aritmético y lógico, desplazamiento reducción
- Operaciones de comparación
- Carga de un valor inmediato

Adicionalmente, algunas de estas operaciones pueden hacerse condicionales.

La YALU soporta: suma, resta, (suma y división por 2), (resta y división por 2).

En modo packed (SIMD, Complex) la XALU puede realizar las siguientes operaciones:

- Suma, resta, (suma y división por 2), (resta y división por 2
- Paso de división
- Valor absoluto
- Desplazamiento aritmético y lógico
- Conjugado
- Negación condicional.

El resultado de estas operaciones es el mismo, sin importar si los datos son SIMD o Complex.

2.5. Buses y Unidades RSS

2.5.1. Xbus e Ybus

Los movimientos entre los registros, e incluso entre memoria y los registros son consecuencia de la arquitectura *load/store* de *CoolFlux Complex.*

CoolFlux Complex posee dos buses centrales para el movimiento de datos, en paralelo con las operaciones de la ruta de datos. Estos buses ofrecen una conexión entre todos los registros, y entre todos los registros y las memorias y dan soporte a los siguientes movimientos de datos:

- Desde un registro fuente a un registro destino.
- Desde un registro fuente a memoria (*store*).
- Desde la memoria a un registro destino (*load*).

Los buses soportan movimientos simples y en paralelo. En un movimiento simple, un solo dato de 24 bits se mueve en un ciclo del procesador. Con un movimiento paralelo, dos datos de 24 bits se mueven en un ciclo del procesador haciendo uso de los dos buses de forma simultanea.

En cada movimiento el valor fuente se convierte a un valor de 24 bits. Este valor de 24 bits se transfiere al registro destino a través de los buses, donde se convierte al tamaño de palabra del destino. La forma en la que se llevan a cabo dichas conversiones depende de si es un movimiento de datos escalares o de datos empaquetados (*Complex*, *SIMD*).

Los buses de 24 bits son suficientes para hacer todos los movimientos de 8-bit, 15-bit, 16-bit y 24-bit de forma directa, pero los movimientos que tienen como fuente o destino un acumulador de 56-bit requiere una atención especial.

2.5.2. Movimientos a un Acumulador

Un acumulador de puede usar como destino de una operación de movimiento de dos maneras:

- Todo el acumulador es el destino (*a0, a1, b0, b1*)
- Un subregistro del acumulador es el destino (*ao0, ao1, ah0, ah1, al0, al1, bo0, bo1, bh0, bh1, bl0, bl1*)

En el caso en el que el destino sea un subregistro, el ancho de la palabra puede ser de 8 bits o de 24 bits. Para un movimiento de un dato de 24 bits al destino, la conversión es directa, se usan los 8 bits menos significativos o no se necesita conversión alguna.

Un acumulador completo puede ser el destino de cuatro maneras:

- Un movimiento de un escalar de 24 bits.
- Un movimiento de un escalar de 48 bits.
- Un movimiento de un dato empaquetado de 24 bits.
- Un movimiento de un dato empaquetado de 48 bits.

En un movimiento de un escalar de 24 bits, el valor de 24 bits se convierte a 56 bits haciendo una extensión de signo en los 8 bits más altos y rellenando con 0 los 24 bits menos significativos. En un movimiento de un escalar de 48 bits, los dos buses (XBUS e YBUS) se combinan para mover un valor de 48 bits. Este valor de 48 bits se convierte a uno de 56 bits, extendiendo el signo en los 8 bits mas altos.

En los movimientos de datos empaquetados de 24 bits, el valor de 24 bits se divide en dos subpalabras de 12 bits. Cuando se convierte a 56 bits, cada una de las partes de 12 bits se convierte a una subpalabra 28 bits. Esto se hace extendiendo el signo de los 4 bits más altos y rellenando con 0 los 12 bits menos significativos. Y los 56 bits del acumulador se rellenan concatenando las dos subpalabras de 28 bits resultantes.

En los movimientos de datos empaquetados de 48 bits, los dos buses se combinan para mover el valor de 48 bits. Este valor de 48 bits ser convierte a 56 bits extendiendo el signo de cada una de las subpalabras de 24 bits con 4 bits.

Los movimientos de 48 bits sólo son posibles entre acumuladores o entre la memoria XY y un acumulador.

2.5.3. Movimientos desde un Acumulador

Un acumulador puede ser usado como fuente de dos maneras:

- La fuente es un acumulador completo (*a0*, *a1*, *b0*, *b1*).
- La fuente es un subregistro del acumulador (*ao0*, *ao1*, *ah0*, *ah1*, *al0*, *al1*, *bo0*, *bo1*, *bh0*, *bh1*, *bl0*, *bl1*).

En el caso en el que un subregistro sea la fuente de un movimiento, el valor es de 8 bits o de 24 bits. La conversión a 24 bits es directa, extensión de signo con 16 bits, o no es necesaria ninguna conversión. La selección del subregistro correcto la realiza la unidad RSS de la que se hablará más adelante.

Un acumulador completo puede ser la fuente de un movimiento de cuatro maneras:

- Un movimiento de un escalar de 24 bits.
- Un movimiento de un escalar de 48 bits.
- Un movimiento de un dato empaquetado de 24 bits.
- Un movimiento de un dato empaquetado de 48 bits.

En el movimiento de un escalar de 24 bits, el valor de 56 bits se convierte a 24 bits a través de la unidad RSS y se transfiere al registro destino utilizando uno de los dos buses.

En el caso de un escalar de 48 bits, los dos buses se combinan para mover el valor de 48 bits en un solo ciclo del procesador. El valor fuente de 56 bits se convierte usando la unidad RSS. Si estamos moviendo un dato empaquetado de 24 bits, los 56 bits de origen se dividen en dos subpalabras de 28 bits que son convertidas a subpalabras de 12 bits gracias a las unidades RSS. Cuando se mueve un dato empaquetado de 48 bits, los dos buses se combinan para poder mover el valor en un solo ciclo del procesador. Los 56 bits del valor de origen pasan a 48 bits utilizando la unidad RSS. En este movimiento, el valor de 56 bits se divide en dos subpalabras de 28 bits y cada una de esas subpalabras de convierte a una subpalabra de 24. Los movimientos que implican datos de 48 bits sólo son posibles entre dos acumuladores o entre un acumulador y la memoria XY.

2.5.4. Unidades RSS

CoolFlux Complex posee dos unidades de redondeo, saturación y selección (RSS), una para los acumuladores A y otra para los acumuladores B. Realizan las conversiones entre los valores de 56 bits de los acumuladores y la precisión requerida para una operación de movimiento (24 bits o 48 bits). Las unidades RSS se utilizan cuando un acumulador de 56 bits se usa como origen en una operación de movimiento, como ya se ha descrito arriba. En el caso en el que se use solo un subregistro de un acumulador, el cometido de las unidades RSS será el de seleccionar la parte correcta (*overflow*, *high*, *low*) y no se realiza conversión alguna.

Si utilizamos los 56 bits del acumulador como fuente, el valor tendrá que ser convertido a 24 bits o a 48 bits, lo que requiere redondeo y saturación. Ambas operaciones (redondeo y saturación) las realiza la unidad RSS que puede operar de varias formas.

Los modos de redondeo son:

- Truncar, bias simple.
- Truncar, signo-magnitud.
- Redondeo, bias simple.
- Redondeo bias balanceado.

Los modos de saturación son:

- Saturación
- Wrapping

El redondeo y la saturación se hace sobre la palabra completa o sobre las dos subpalabras, dependiendo si estamos operando en modo escalar o empaquetado. En ambos casos tanto el redondeo como la saturación funcionan de la misma manera.

Las dos unidades RSS siempre funcionan del mismo modo, y se configura mediante los bits *sr.b*,*sr.r*y *sr.s* del registro de estado.

2.6. Memorias y E/S

La arquitectura de *CoolFlux Complex* es una arquitectura dual Harvard y tiene dos memorias de datos separadas X e Y. Cada una de las memorias tiene 16 Mpalabras de 24 bits. En un solo ciclo del procesador cada memoria puede ser accedida de forma separada, y esto permite que dos datos puedan ser leídos o escritos en un solo ciclo. También existe un modo especial en el que las dos memorias pueden combinarse de forma lógica en una única memoria que puede almacenar datos de 48 bits.

De una manera similar a las memorias de datos, la E/S de *CoolFlux Complex* está mapeada en memoria. El espacio de E/S tiene también 16 Mpalabras de 24 bits. Desde el punto de vista del software, la E/S se comporta como una memoria en el que los datos se pueden leer y escribir.

2.6.1. Memorias de Datos Individuales (X, Y)

CoolFlux Complex tiene dos memorias de datos separadas de 16 Mpalabras de 24 bits. Normalmente los accesos a las memorias se hacen en paralelo para poder tener un ancho de banda suficiente para el cómputo que se esté realizando.

2.6.2. Memoria de Doble Precisión (XY)

Hay instrucciones de load y store de doble precisión, que pueden mover un solo dato de 48 bits desde/al subsistema de memoria. Para estas instrucciones ambas memorias X e Y, y los dos buses se combinan. En este modo las dos memorias reciben las mismas direcciones, es por eso que es necesario reservar las posiciones correspondientes en ambas memorias. La memoria XY usa la unidad de direccionamiento X y tiene las mismas capacidades de direccionamiento que la memoria X.

2.6.3. Espacio de Memoria E/S

La E/S de *CoolFlux Complex* esta mapeada en memoria. El espacio de E/S es de 16 Mpalabras de 24 bits de ancho. El espacio de memoria E/S puede ser direccionado del mismo modo que el espacio de memoria Y, ya que se utiliza la unidad de direccionamiento Y para generar sus direcciones.

2.7. Unidades de Direccionamiento X e Y

Las unidades de direccionamiento X e Y generan las direcciones para los accesos a memoria. Estas unidades soportan los siguientes modos de direccionamiento:

- Direccionamiento directo.
- Direccionamiento indirecto con post-modificación
- Direccionamiento indexado en el puntero de la pila.

El soporte del direccionamiento indirecto implica la existencia de dos bancos de registros puntero: XPTR, YPTR. Están ubicados en los unidades de direccionamiento. El banco de registros puntero tiene 8 punteros (*xptr0... xptr7*); el banco YPTR tiene 4 punteros (*yptr0… yptr3*). Cuando se accede a las memorias indirectamente haciendo uso de estos punteros, el valor del puntero puede ser actualizado (post-modificado) en paralelo con el acceso. Los cálculos para esta actualización se realizan en la XAGU e YAGU, las unidades de generación de direcciones (*AdressGenerationUnit*). La operación de post-modificación se especifica como parte del acceso a memoria. Estas operaciones usan los registros de otros bancos de registros: XSTEP, YSTEP, XMOD e YMOD. Las posibles operaciones de post-modificación son:

- Nop (no operación)
- Incremento
- Decremento
- Incrementar en 2
- Decrementar en 2
- Suma de un registro *step*
- Resta de un registro *step*
- Y un modo especial que se usa para
    - Direccionamiento de buffer circular
    - Direccionamiento de *bit-reverse*

Aparte de las operaciones de actualización que son parte de la sintaxis del direccionamiento indirecto, existen otras operaciones de punteros específicos que se ejecutan en la XAGU y en la YAGU. Dichas operaciones son ejecutadas en paralelo con las operaciones de movimiento en los buses Xbus e Ybus, accesos a memoria, y las operaciones de la ruta de datos (XMUL, YMUL, XALU e YALU).

2.8. Unidad de Control de Programa

Esta unidad controla el contador de programa, es registro de estado y los registros utilizados para interrupciones y saltos a subrutina. Controla el flujo de programa e implementa saltos, llamadas a subrutina, bucles hardware y maneja interrupciones. Además, implementa cuatro registros que son directamente accesibles desde el código:

Nombre	Nombre largo	Tamaño	Descripción
sr	Registro de estado	24 bits	Contiene la palabra del estado y control de la maquina.
lr	Registro de enlace	24 bits	Contiene la dirección de retorno después de una instrucción *call*.
ilr	Registro de enlace de interrupción	24 bits	Contiene la dirección de retorno después de una interrupción
isr	Registro de estado de interrupción	24 bits	Se usa para salvar el valor del registro *sr* durante una interrupción.

Tabla 1: Registros de la Unidad de Control de Programa

El registro de estado contiene tres *flags* (*zero*, *negative* y *overflow*) que son activados por la XALU. Hay varios bits de control que controlan el modo de operación de *CoolFlux Complex*. Tres bits controlan el modo de la unidad RSS (*sr.b*, *sr.r, sr.s*). El bit *sr.ie* es el bit de activación de las interrupciones. Dos bits se usan para la división de datos empaquetados (*sr.div0*, *sr.div1*). Dos bits controlan que no se produzcan desbordamientos en el movimiento de datos empaquetados, uno (sr.ov_i) para la parte imaginaria (o la primera subpalaba) y el otro (sr.ov_r) para la parte real (o la segunda subpalabra).Un último bit sirve para diferenciar entre las operaciones con complejos o con SIMD, aunque la única diferencia entre ambos es en la multiplicación.

2.9. Pila de Bucles Hardware

Los bucles hardware se implementan usando la pila de bucles hardware. La pila tiene cuatro niveles de profundidad, es decir, el hardware soporta hasta cuatro bucles anidados. Cuando de inicia un bucle, se meten en la pila las direcciones de comienzo y fin del bucle, así como el número de veces que el bucle ha de ejecutarse. Cuando el bucle finaliza, esta información se saca de la pila que se compone de:

- Cuatro direcciones de comienzo de 24 bits.
- Cuatro direcciones de fin de 24 bits.
- Cuatro contadores de bucle de 24 bits (hasta 16.777.215 iteraciones)

3. Los Tipos SIMD

Lo que diferencia *CoolFlux Complex* de sus anteriores versiones es que, además de los tipos básicos de datos, incorpora los siguientes tipos de datos empaquetados:

Nombre	Tipo	Nº de bits	Nº de bits por componente	Rango por componente
complex12	complex	24	12	$[-1, 1-2^{-11}]$
complex24	complex	48	24	$[-1, 1-2^{-23}]$
complex28	complex	56	28	$[-2^{3}, 2^{3}-2^{-23}]$
simd12	SIMD	24	12	$[-1, 1-2^{-11}]$
simd24	SIMD	48	24	$[-1, 1-2^{-23}]$
simd28	SIMD	56	28	$[-2^{3}, 2^{3}-2^{-23}]$

Tabla 2: Tipos de datos empaquetados.

Estos nuevos tipos de datos tienen dos componentes del mismo tamaño: componente real e imaginario para los tipos *complex*, y componentes alto y bajo en el caso de los tipos *SIMD*. Tanto el componente real como el bajo están ubicados en los bits menos significativos de la palabra. Los tipos *complex12*, *complex24*, *simd12* y *simd24* tienen un punto binario a la derecha del bit más significativo de cada componente. Los *complex28* y *simd28* corresponden a los acumuladores, cada componente de estos tipos tiene 4 bits de *overflow*, por lo tanto el punto binario está situado a la derecha el quito bit más significativo.

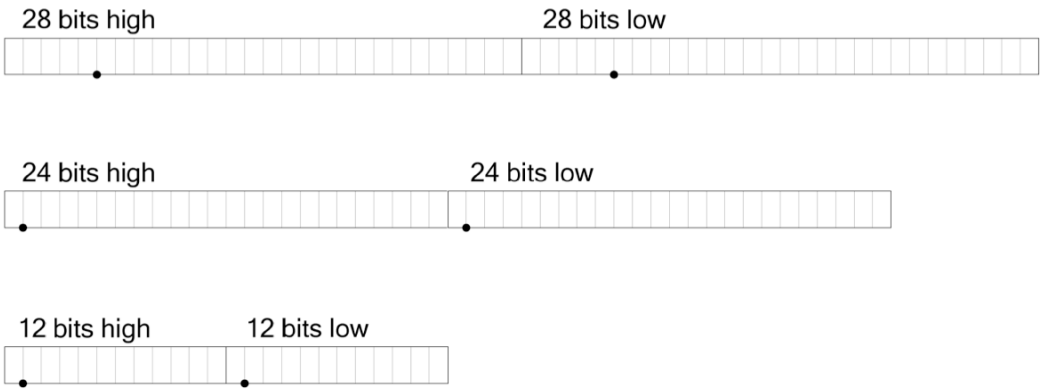

Ilustración 3: Los tres tipos de datos SIMD (simd28, simd24, simd12).

Vamos a ver, los operadores C que pueden ser utilizados con los tipos SIMD y de qué manera se comportan.

Tipo	Operadores disponibles
simd12	todos los operadores C excepto /, %, ++ y --
simd24	todos los operadores C excepto /, %, ++ y --
simd28	todos los operadores C excepto /, %, ++ y --

Tabla 3: Operadores C para SIMD

Cuando se suman dos valores simd12, se asume que no va a haber desbordamiento, si esto ocurre, el valor resultante se considera indefinido.

Para usar los bits de overflow del acumulador, un hay que realizar un cast explícito al tipo simd28. Cuando el resultado vuelva a ser asignado a un valor simd12 o simd24, pasará por la unidad RSS.

Los operadores de desplazamiento (<< y >>) desplazan cada componente de su primer operando a la izquierda o a la derecha, respectivamente, el número de bits especificado por el segundo operando. El segundo operando debe ser:

- Un entero positivo.
- Menor o igual al número de bits de los componentes del primer operando.

La división (/) y el módulo (%) tiene soporte para las variables simd12 y simd28. El bucle principal para una operación de 12 bits tarda en ejecutarse como mucho 12 ciclos. A esto se le añaden algunos ciclos extra para chequear los signos de los operandos.

Algunas cosas a tener en cuenta cuando estamos trabajando con datos *SIMD* es que la división por cero, no produce un bucle infinito, pero produce resultados erróneos. simd28 operator/ (simd12 a, int b) divide cada uno de los componentes por un entero b. Algunas operaciones que no pueden llevarse a cabo con los operadores, están soportadas por funciones.

Nombre de la Función	Descripción
simd12 cneg (simd12 a, simd12 b)	Niega el componente de a cuando el correspondiente componente de b es negativo.
simd28 cneg (simd28 a, simd28 b)	Niega el componente de a cuando el correspondiente componente de b es negativo.
simd12 abs (simd12)	Toma el valor absoluto de cada uno de los componentes.
simd28 abs (simd28)	Toma el valor absoluto de cada uno de los componentes.
simd12 join_simd12 (fix l, fix h)	Une los componentes l e h en un tipo empaquetado.
simd24 join_simd24 (fix l, fix h)	Une los componentes l e h en un tipo empaquetado.
simd28 join_simd28 (fix l, fix h)	Une los componentes l e h en un tipo empaquetado.
acc extract_high (simd12)	Extrae el componente alto (bits 23..12) y lo alinea con el punto binario.
acc extract_low (simd12)	Extrae el componente bajo (bits 11..0) y lo alinea con el punto binario.
acc extract_high (simd24)	Extrae el componente alto (bits 47..24) y lo alinea con el punto binario.
acc extract_low (simd24)	Extrae el componente bajo (bits 23..0) y lo alinea con el punto binario.
acc extract_high (simd28)	Extrae el componente alto (bits 55..28) y lo alinea con el punto binario.
acc extract_low (simd28)	Extrae el componente bajo (bits 27..0) y lo alinea con el punto binario.

Tabla 4: Funciones SIMD

4. Trabajo Realizado

Antes de comenzar con la optimización en sí, se escribieron las funciones necesarias en la iolib, que es la librería que nos permite leer valores de memoria de cualquier tipo soportado por *CoolFlux Complex*. Dado que el procesador era nuevo, esta librería aún no estaba terminada y antes de todo, se implementaron las funciones que se encargaban de leer y escribir los tipos de datos SIMD, que son en los que se centra este trabajo.

Para la realización del trabajo se disponía de diferentes herramientas y entornos de programación. El primer entorno utilizado es el Microsoft Visual Studio 6, este es el entorno donde todo empieza. Lo primero es hacer las funciones en este entorno, y gracias a las librerías nativas se sabe si la función compila sin problemas. El siguiente paso consiste en preparar un pequeño programa que utiliza la nueva función. Este pequeño programa toma como entrada una serie de valores para los cuales se conoce los resultados de antemano. Después se realiza una comparación de los resultados obtenidos y los conocidos para poder validarlos. Una vez que la función escrita en C funciona sin problemas con la librería nativa, se compila el código con un compilador propio de *CoolFlux Complex* llamado Chess. Este compilador traduce el código C a código ensamblador de *CoolFlux Complex*, dicho compilador produce un código muy optimizado.

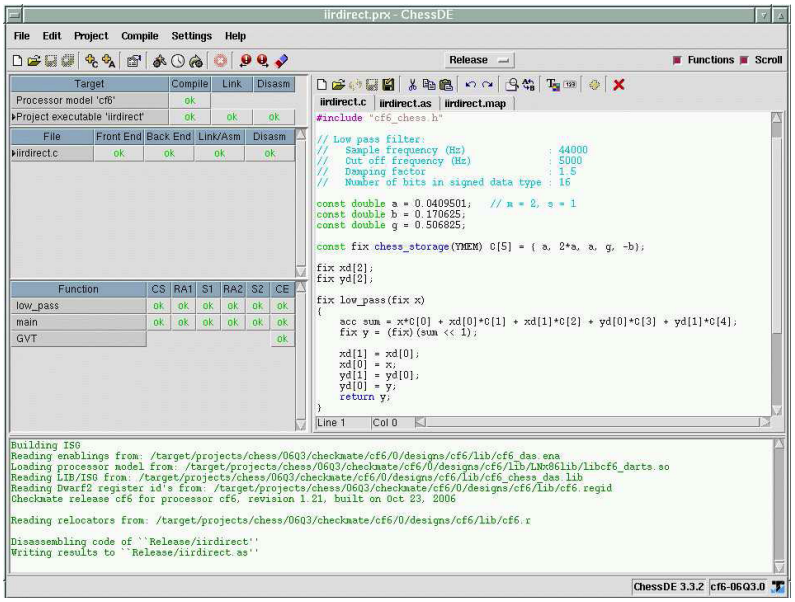

Ilustración 4: Interfaz de usuario de Chess

Una vez tenemos el código ensamblador, podemos utilizar un simulador de *CoolFlux Complex*, llamado ISS, y ejecutar ese código. El simulador nos permite ejecutar paso a paso ese código y observar el contenido de las memorias, el contenido de los registros, el número de ciclos que han pasado desde el inicio de la ejecución y muchos más datos que nos ayudan a decidir cuánto de buena es la optimización realizada.

El simulador, también nos permite mapear archivos de texto en la memoria de entrada/salida del DSP, lo que nos permite ejecutar sobre el simulador las mismas pruebas que se realizan con las librerías nativas. El resultado de ambas pruebas se comparan entre sí. Si el resultado de la comparación nos indica que los resultados son diferentes, se inicia el proceso de depuración que determina si la discrepancia se debe a un error en la librería nativa o en el procesador.

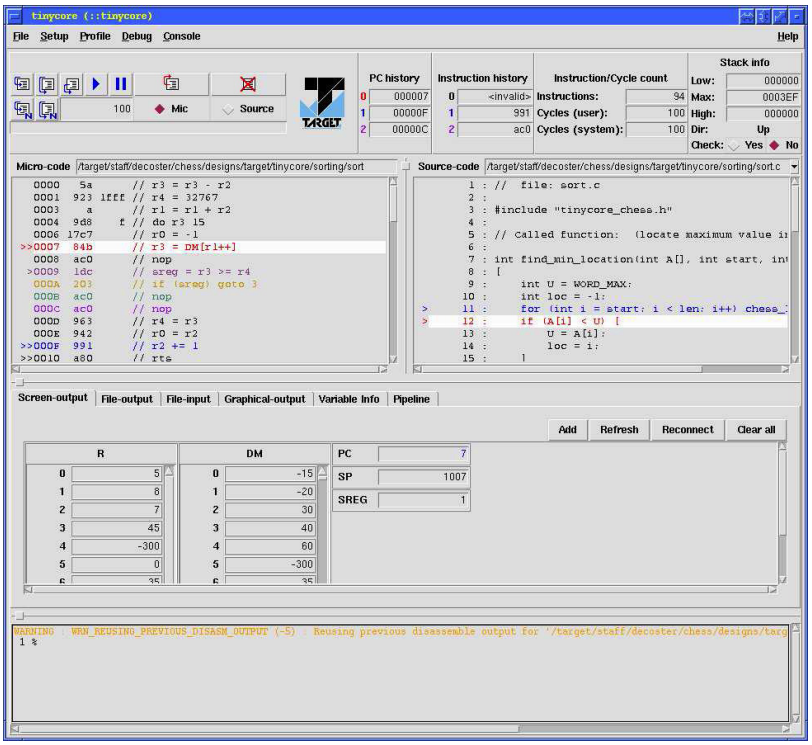

Ilustración 5: Interfaz de usuario de ISS

Una vez que los resultados de la ejecución con Visual Studio y el simulador dan como resultado los mismos valores, y estos a su vez son iguales a los esperados, la función queda validada y se supone correcta. Este proceso se hace tantas veces sea necesario hasta obtener una función correcta y que consuma el mínimo número de ciclos posible.

4.1. Filtro FIR

FIR es un acrónimo en inglés para Finite Impulse Response o Respuesta finita al impulso. Se trata de un tipo de filtros digitales en el que, como su nombre indica, si la entrada es una señal impulso, la salida tendrá un número finito de términos no nulos.

La estructura básica de un FIR es:

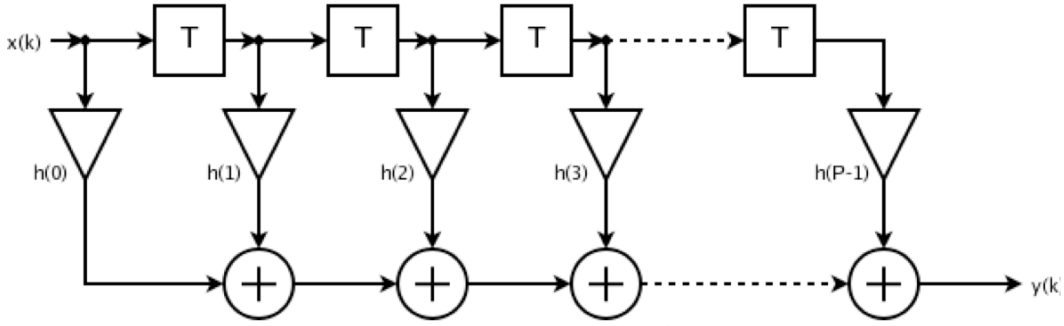

Ilustración 6: Estructura básica filtro FIR

Pueden hacerse multitud de variaciones de esta estructura. Hacerlo como varios filtros en serie, en cascada, etc.

La expresión matemática del filtro FIR es la siguiente:

$$y_n = \sum_{k=0}^{N-1} b_k x(n-k)$$

Al término que esta dentro del sumatorio lo llamaremos tap, es decir, cuando se haga referencia a un tap del filtro, se estará haciendo referencia al proceso de una multiplicación y una suma, equivalente a una operación MAC de *CoolFlux Complex*.

Las funciones a optimizar son las siguientes:

- Filtro FIR basado en muestras: Recibe una muestra, y devuelve como salida la muestra procesada.
- Filtro FIR basado en bloques: Recibe un bloque de muestras y devuelve un bloque, de la misma longitud, de muestras ya procesadas.
- Filtro FIR basado en bloques estéreo: Recibe un bloque de muestras pertenecientes a dos canales entrelazados y lo devuelve después de procesarlas.

4.1.1. FIR Basado en Muestras

Esta es la forma en la que el *CoolFlux* realiza un filtro FIR basado en muestras, utilizando el tipo de datos fix. La función, calcula una sola salida, es la que se corresponde con la muestra de entrada. En la explicación vamos a tomar un filtro de longitud 64. Cada rectángulo es equivalente a una palabra de memoria (24 bits).

Vector de coeficientes de tamaño 64 (64 palabras):

h_0	h_1	h_2	h_3	...	h_{63}

Vector de retardos de tamaño 63 (63 palabras):

X_{n-1}	X_{n-2}	X_{n-3}	...	X_{n-63}

Muestra de entrada (1 palabra):

X_n

Cómputo de una muestra de salida:

Y_n	=	X_n	*	h_0	+	X_{n-1}	*	h_1	+	...	+	X_{n-63}	*	h_{63}

Nuevo vector de retardos:

X_n	X_{n-1}	X_{n-2}	...	X_{n-62}

Antes de hacer la llamada, por primera vez, a la función es necesario inicializar sus posiciones con el valor cero. Además, el vector ha de estar alineado en memoria, siguiendo las restricciones impuestas por el *CoolFlux Complex* para poder usar ese vector como un vector circular.

El vector de coeficientes y el vector de retardos han de estar ubicados en memorias diferentes con el fin de poder acceder a ambos vectores en un solo ciclo. Por ello, el vector de retardos estará ubicado en la memoria X, y el vector de coeficientes estará en la memoria Y.

La arquitectura de *CoolFlux Complex* nos permite realizar en un mismo dos accesos a memoria y una operación MAC, pero en esa operación no es posible utilizar los datos leídos en el mismo ciclo. Con

el fin de poder conseguir el procesado de un tap por ciclo, deberemos hacer el calculo de un tap y la lectura de datos del siguiente.

Ahora veamos qué pasa si utilizamos el tipo de datos SIMD, con el cual, en una sola palabra de memoria tenemos almacenados dos datos.

La forma de representar los datos de tipo SIMD será la siguiente: cada rectángulo representará una palabra de memoria (como en el caso del tipo fix) pero ésta vez, dicho rectángulo estará dividido horizontalmente por una línea discontinua que hará de separación entre los dos datos que contiene cada palabra de tipo SIMD.

Ejemplo:

low	Bits menos significativos de la palabra.
high	Bits más significativos de la palabra.

En la primera aproximación, vamos a ver qué pasaría si utilizamos el mismo algoritmo que utilizamos para el tipo de datos *fix*.

Vector de coeficientes de tamaño 64 (32 palabras):

h_0	h_2	h_4	h_6	...	h_{62}
h_1	h_3	h_5	h_7	...	h_{63}

Vector de retardos de tamaño 62 (31 palabras):

X_{n-2}	X_{n-4}	X_{n-6}	...	X_{n-62}
X_{n-3}	X_{n-5}	X_{n-7}	...	X_{n-63}

Muestra de entrada (1 palabra):

X_n
X_{n-1}

Cómputo de una muestra de salida:

$$\begin{bmatrix} Y_n \\ Y_{n-1} \end{bmatrix} = \begin{bmatrix} X_n \\ X_{n-1} \end{bmatrix} * \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} + \begin{bmatrix} X_{n-2} \\ X_{n-3} \end{bmatrix} * \begin{bmatrix} h_2 \\ h_3 \end{bmatrix} + \ldots + \begin{bmatrix} X_{n-62} \\ X_{n-63} \end{bmatrix} * \begin{bmatrix} h_{62} \\ h_{63} \end{bmatrix}$$

Nuevo vector de retardos:

X_n	X_{n-2}	X_{n-4}	...	X_{n-60}
X_{n-1}	X_{n-3}	X_{n-5}	...	X_{n-61}

A la vista de los valores que toman Yn e Yn+1, se pueden sacar varias conclusiones:

- Yn no está bien calculado.
- Yn-1 no está bien calculado.
- La salida Yn que buscamos es la suma de low + high del resultado que hemos obtenido.

Después de las esas conclusiones, ya sabemos cómo calcular Yn, pero seguimos teniendo un problema, Yn-1 se queda sin calcular.

En éste punto tenemos un filtro FIR que opera un el tipo de datos SIMD, pero sólo nos calcula una de cada dos muestras. La solución que propongo es la siguiente: tener otro vector de coeficientes igual al anterior pero desplazado 12 bits, así con el vector de coeficientes original haremos el cálculo de Yn y con el vector de coeficientes desplazado haremos el cálculo de Yn-1.

Ésta es la segunda aproximación al cálculo de un filtro FIR basado en muestras utilizando el tipo de datos SIMD.

Vector de coeficientes de tamaño 64 (32 palabras):

h_0	h_2	h_4	h_6	...	h_{62}
h_1	h_3	h_5	h_7	...	h_{63}

Vector de coeficientes desplazado de tamaño 66 (33 palabras):

0	h_1	h_3	h_5	...	h_{61}	h_{63}
h_0	h_2	h_4	h_6	...	h_{62}	0

Vector de retardos de tamaño 62 (31 palabras):

X_{n-2}	X_{n-4}	X_{n-6}	...	X_{n-62}
X_{n-3}	X_{n-5}	X_{n-7}	...	X_{n-63}

Muestra de entrada (1 palabra):

X_n
X_{n-1}

Cómputo de una muestra de salida:

$$\begin{bmatrix} Y_{n(lo)} \\ Y_{n(hi)} \end{bmatrix} = \begin{bmatrix} X_n \\ X_{n-1} \end{bmatrix} * \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} + \begin{bmatrix} X_{n-2} \\ X_{n-3} \end{bmatrix} * \begin{bmatrix} h_2 \\ h_3 \end{bmatrix} + \dots + \begin{bmatrix} X_{n-62} \\ X_{n-63} \end{bmatrix} * \begin{bmatrix} h_{62} \\ h_{63} \end{bmatrix}$$

$$\begin{bmatrix} Y_{n-1(lo)} \\ Y_{n-1(hi)} \end{bmatrix} = \begin{bmatrix} X_n \\ X_{n-1} \end{bmatrix} * \begin{bmatrix} 0 \\ h_0 \end{bmatrix} + \begin{bmatrix} X_{n-2} \\ X_{n-3} \end{bmatrix} * \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + \dots + \begin{bmatrix} ¿? \\ ¿? \end{bmatrix} * \begin{bmatrix} h_{63} \\ 0 \end{bmatrix}$$

Y_n	Yn = Yn(lo)+ Yn(hi)
Y_{n-1}	Yn-1 = Yn-1(lo) + Yn-1(hi)

Nuevo vector de retardos:

X_n	X_{n-2}	X_{n-4}	...	X_{n-60}
X_{n-1}	X_{n-3}	X_{n-5}	...	X_{n-61}

En esta aproximación se han resuelto la mayoría de los problemas que surgían en la anterior. Sin embargo, el cálculo no es correcto. Esto se debe a que el vector de retardos no almacena los suficientes retardos para poder completar el cálculo de Yn-1.

La siguiente manera de hacer el cálculo del filtro FIR basado en muestras, soluciona todos los problemas anteriormente encontrados.

Vector de coeficientes de tamaño 64 (32 palabras):

h_0	h_2	h_4	h_6	...	h_{62}
h_1	h_3	h_5	h_7	...	h_{63}

Vector de coeficientes desplazado de tamaño 66 (33 palabras) :

0	h_1	h_3	h_5	...	h_{61}	h_{63}
h_0	h_2	h_4	h_6	...	h_{62}	0

Vector de retardos de tamaño 64 (32 palabras):

X_{n-2}	X_{n-4}	X_{n-6}	...	X_{n-62}	X_{n-64}
X_{n-3}	X_{n-5}	X_{n-7}	...	X_{n-63}	X_{n-65}

Muestra de entrada (1 palabra):

X_n
X_{n-1}

Cómputo de una muestra de salida:

$$\begin{bmatrix} Y_{n(lo)} \\ Y_{n(hi)} \end{bmatrix} = \begin{bmatrix} X_n \\ X_{n-1} \end{bmatrix} * \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} + \begin{bmatrix} X_{n-2} \\ X_{n-3} \end{bmatrix} * \begin{bmatrix} h_2 \\ h_3 \end{bmatrix} + \ldots + \begin{bmatrix} X_{n-62} \\ X_{n-63} \end{bmatrix} * \begin{bmatrix} h_{62} \\ h_{63} \end{bmatrix}$$

$$\begin{bmatrix} Y_{n-1(lo)} \\ Y_{n-1(hi)} \end{bmatrix} = \begin{bmatrix} X_n \\ X_{n-1} \end{bmatrix} * \begin{bmatrix} 0 \\ h_0 \end{bmatrix} + \begin{bmatrix} X_{n-2} \\ X_{n-3} \end{bmatrix} * \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + \ldots + \begin{bmatrix} X_{n-64} \\ X_{n-65} \end{bmatrix} * \begin{bmatrix} h_{63} \\ 0 \end{bmatrix}$$

Y_n	Yn = Yn(lo)+ Yn(hi)
Y_{n-1}	Yn-1 = Yn-1(lo) + Yn-1(hi)

Nuevo vector de retardos:

X_n	X_{n-2}	X_{n-4}	...	X_{n-60}	X_{n-62}
X_{n-1}	X_{n-3}	X_{n-5}	...	X_{n-61}	X_{n-63}

Ya tenemos un algoritmo que nos permite calcular el filtro FIR basado en muestras utilizando datos del tipo *SIMD*.

Para un filtro de longitud 64, la versión que trabaja sobre muestras *SIMD* necesitaría 84 ciclos para procesar dicha muestra, mientras que la versión que trabaja sobre muestras fix necesitaría 75 ciclos.

Si nos fijamos en el número de ciclos por llamada a la función, la nueva versión del filtro es más lenta, pero hay que tener en cuenta que la versión *SIMD*, realmente está procesando dos muestras al mismo tiempo. Por lo que podemos ver que con sólo 9 ciclos más, la versión *SIMD* procesa el doble. En ambos casos, el número de multiplicaciones es el mismo, pero al utilizar instrucciones *SIMD* procesamos en paralelo dos muestras. El rendimiento no es exactamente el doble,

porque el manejo de datos *SIMD* tiene mayor número de instrucciones de control.

Ahora tenemos una función que calcula un filtro FIR haciendo el cálculo por muestras que necesita dos muestras, cuando lo más lógico sería que la entrada fuera una única muestra. Tras está función hice una nueva, que recibe una muestra del tipo *fix* la procesa en forma *SIMD* y devuelve un resultado del tipo *fix*.

En ésta nueva función, vamos a trabajar con operaciones *SIMD* por lo que vamos a seguir teniendo dos vectores de coeficientes y la línea de retardo del tipo *SIMD*. La novedad en esta función es que al procesar una sola muestra, la línea de retardo va ha poder estar en dos situaciones diferentes. En una de ellas, la muestra habrá que juntarla con un 0 y utilizar los coeficientes desplazados, y en la otra, la muestra habrá que juntarla con la muestra anterior y utilizar el vector de coeficientes sin desplazamiento. Para saber en cuál de las dos situaciones estamos, introduciremos un nuevo parámetro que llamaremos joinPos. Ahora veremos el algoritmo de una forma más detallada.

Ésta es la forma de procesar una muestra cuando joinPos = 0:

Vector de coeficientes desplazado de tamaño 66 (33 palabras):

0	h_1	h_3	h_5	...	h_{61}	h_{63}
h_0	h_2	h_4	h_6	...	h_{62}	0

Vector de retardos de tamaño 64 (32 palabras):

X_{n-1}	X_{n-3}	X_{n-5}	X_{n-7}	...	X_{n-63}
X_{n-2}	X_{n-4}	X_{n-6}	X_{n-8}	...	X_{n-64}

Muestra de entrada (1 palabra):

Xn

Cómputo de una muestra de salida:

Muestra de entrada (*SIMD*)

0
X_n

Calculamos el resultado utilizando la nueva muestra *SIMD* y los vectores de retardo y coeficientes.

$$\begin{bmatrix} Y_{n(lo)} \\ Y_{n(hi)} \end{bmatrix} = \begin{bmatrix} 0 \\ X_n \end{bmatrix} * \begin{bmatrix} 0 \\ h_0 \end{bmatrix} + \begin{bmatrix} X_{n-1} \\ X_{n-2} \end{bmatrix} * \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + \ \ldots \ + \begin{bmatrix} X_{n-63} \\ X_{n-64} \end{bmatrix} * \begin{bmatrix} h_{63} \\ 0 \end{bmatrix}$$

Introducimos la nueva muestra, en el vector de retardos.

0	X_{n-1}	X_{n-3}	X_{n-5}	...	X_{n-61}
X_n	X_{n-2}	X_{n-4}	X_{n-6}	...	X_{n-62}

joinPos = 1 – joinPos

Y_n Yn = Yn(lo)+ Yn(hi)

Ésta es la forma de procesar una muestra cuando joinPos = 1:

Vector de coeficientes de tamaño 64 (32 palabras):

h_0	h_2	h_4	h_6	...	h_{62}
h_1	h_3	h_5	h_7	...	h_{63}

Vector de retardos de tamaño 64 (32 palabras):

0	X_{n-2}	X_{n-4}	X_{n-6}	...	X_{n-62}
X_{n-1}	X_{n-3}	X_{n-5}	X_{n-7}	...	X_{n-63}

Muestra de entrada (1 palabra):

X_n

Cómputo de una muestra de salida:

Introducimos la nueva muestra, en la primera posición del vector de retardos, donde antes había un cero.

X_n	X_{n-2}	X_{n-4}	X_{n-6}	...	X_{n-62}
X_{n-1}	X_{n-3}	X_{n-5}	X_{n-7}	...	X_{n-63}

Calculamos el resultado utilizando los vectores de retardo y coeficientes.

$$\begin{bmatrix} Y_{n(lo)} \\ Y_{n(hi)} \end{bmatrix} = \begin{bmatrix} X_n \\ X_{n-1} \end{bmatrix} * \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} + \begin{bmatrix} X_{n-2} \\ X_{n-3} \end{bmatrix} * \begin{bmatrix} h_2 \\ h_3 \end{bmatrix} + \ \ldots \ + \begin{bmatrix} X_{n-62} \\ X_{n-63} \end{bmatrix} * \begin{bmatrix} h_{62} \\ h_{63} \end{bmatrix}$$

joinPos = 1 – joinPos

Y_n	$Y_n = Y_{n(lo)} + Y_{n(hi)}$

Como se puede ver en el esquema del algoritmo, es necesario que el vector de retardos sea de longitud *n*, siendo *n* el número de coeficientes del filtro. Por lo demás, el algoritmo es muy parecido al que procesaba dos muestras en *SIMD*. Sin embargo, el rendimiento es algo menor debido a la mayor carga de control requerida para el paso de *fix* a *SIMD*.

El número de ciclos necesarios en cada llamada a cada una de las funciones vistas:

Nombre de la Función	Número de Ciclos
fir_1ch_sb_Process(simd)	24 + (FilterLength/2 – 2) * 2
fir_1ch_sb_Process(fix)	13 + (FilterLength-2)
fir_1ch_sb_Process(simd-fix)	25 + (FilterLength/2 – joinPos)

Tabla 5: Rendimiento FIR basado en muestras

Se observa que la nueva función es casi tan buena como la anterior, de hecho, por cada llamada a la función, se necesitan algo más de la mitad de ciclos. Pero en este caso sólo se procesa una muestra.

La siguiente gráfica representa el número de ciclos necesarios para calcular una muestra en cada una de las versiones del filtro vistas hasta ahora, para diferentes valores de Filterlength.

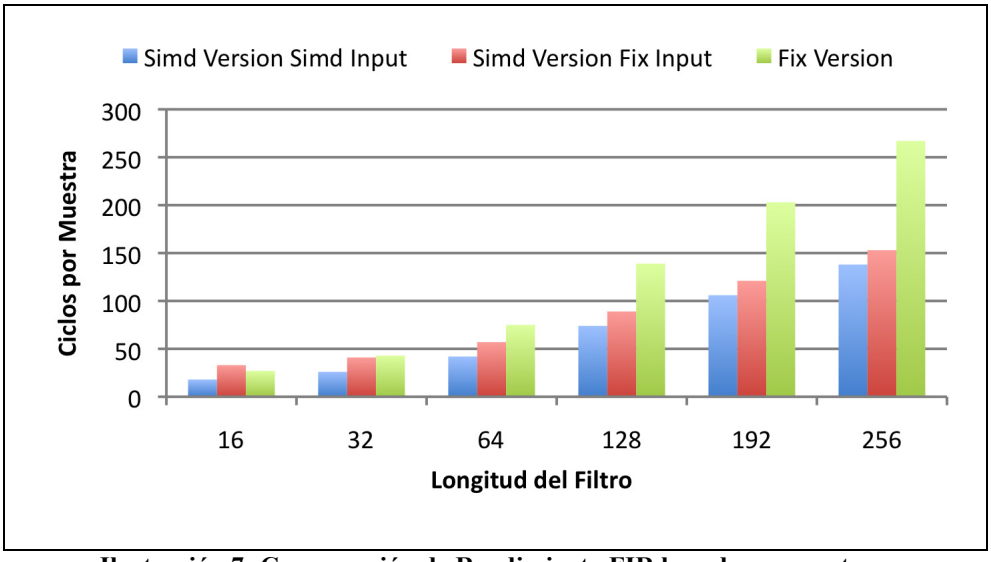

Ilustración 7: Comparación de Rendimiento FIR basado en muestras.

4.1.2. **FIR Basado en Bloques de Muestras**

Vamos a ver cómo el *CoolFlux Complex* hace el filtro FIR basado en bloques utilizando el tipo de datos fix. Para la explicación vamos a tomar un filtro de longitud 64 y un bloque de 128 muestras.

Vector de coeficientes de tamaño 64 (64 palabras):

h_0	h_1	h_2	h_3	...	h_{63}

Vector de retardos de tamaño 63 (63 palabras):

X_{n-1}	X_{n-2}	X_{n-3}	...	X_{n-63}

Bloque de entrada (128 palabras):

X_0	X_1	X_2	X_3	...	X_{127}

Cómputo de una muestra de salida:

$$Y_n = X_n * h_0 + X_{n-1} * h_1 + \ldots + X_{n-63} * h_{63}$$

Vector de retardos después de computar el bloque:

X_{127}	X_{126}	X_{125}	...	X_{65}

Vector de salida:

Y_0	Y_1	Y_2	Y_3	...	Y_{127}

El algoritmo como se puede ver, es el mismo que en el caso de la función que calcula el filtro muestra a muestra. Pero, al disponer de varias muestras, podemos hacer el cálculo de dos muestras simultáneamente al disponer de dos unidades MAC en el *CoolFlux Complex*. Para poder hacer el mayor número de operaciones por ciclo en la ejecución del algoritmo, habrá que introducir, en la función, una serie de precondiciones que ya veremos más adelante junto con el código correspondiente.

Ahora veamos de qué manera podemos optimizar el algoritmo usando datos del tipo *SIMD*, que se basa en el algoritmo utilizado para la versión que procesaba muestras unitarias del tipo *SIMD*:

Vector de coeficientes de tamaño 64 (32 palabras):

h_0	h_2	h_4	h_6	...	h_{62}
h_1	h_3	h_5	h_7	...	h_{63}

Vector de coeficientes desplazado de tamaño 66 (33 palabras):

0	h_1	h_3	h_5	...	h_{61}	h_{63}
h_0	h_2	h_4	h_6	...	h_{62}	0

Vector de retardos de tamaño 64 (32 palabras):

X_{n-2}	X_{n-4}	X_{n-6}	...	X_{n-62}	X_{n-64}
X_{n-3}	X_{n-5}	X_{n-7}	...	X_{n-63}	X_{n-65}

Bloque de entrada (64 palabras):

X_1	X_3	X_5	X_7	...	X_{127}
X_0	X_2	X_4	X_6	...	X_{126}

Cómputo de una muestra de salida:

$$\begin{bmatrix} Y_{n(lo)} \\ Y_{n(hi)} \end{bmatrix} = \begin{bmatrix} X_n \\ X_{n-1} \end{bmatrix} * \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} + \begin{bmatrix} X_{n-2} \\ X_{n-3} \end{bmatrix} * \begin{bmatrix} h_2 \\ h_3 \end{bmatrix} + \ldots + \begin{bmatrix} X_{n-62} \\ X_{n-63} \end{bmatrix} * \begin{bmatrix} h_{62} \\ h_{63} \end{bmatrix}$$

$$\begin{bmatrix} Y_{n-1(lo)} \\ Y_{n-1(hi)} \end{bmatrix} = \begin{bmatrix} X_n \\ X_{n-1} \end{bmatrix} * \begin{bmatrix} 0 \\ h_0 \end{bmatrix} + \begin{bmatrix} X_{n-2} \\ X_{n-3} \end{bmatrix} * \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + \ldots + \begin{bmatrix} X_{n-64} \\ X_{n-65} \end{bmatrix} * \begin{bmatrix} h_{63} \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} Y_n \\ Y_{n-1} \end{bmatrix} \quad \begin{matrix} Y_n = Y_{n(lo)} + Y_{n(hi)} \\ Y_{n-1} = Y_{n-1(lo)} + Y_{n-1(hi)} \end{matrix}$$

Vector de retardos después de computar el bloque

X_{127}	X_{125}	X_{123}	...	X_{65}	X_{63}
X_{126}	X_{124}	X_{122}	...	X_{64}	X_{62}

Viendo cómo trabajan ambos algoritmos, parece que la ganancia teórica sería del 100%, pero esto no es así. La causa que hace esto imposible, es que estamos trabajando con dos vectores de coeficientes en vez de uno. El trabajar con un vector más, implica que en lugar de poder hacer el bucle mas interno en dos ciclos, se hagan en tres.

El número mínimo de elementos que debe tener el bloque, para que el la función pueda procesarlo de forma correcta, debe de ser par puesto que estamos utilizando dos muestras al mismo tiempo. También es importante darse cuenta del número de valores qué realmente hacen falta son cuatro, teniendo en cuenta que cada muestra SIMD contiene dos valores.

Las siguientes ecuaciones indican el número de ciclos necesarios para la ejecución de una llamada a los procedimientos:

Nombre de la Función	Número de Ciclos
fir_1ch_bb_Process(simd)	16 + (BlockLength/4) * (47+((FilterLength/2) – 2) * 3)
fir_1ch_bb_Process (fix)	14 + (BlockLength/2) * (27+((FilterLength/2) – 2) * 2)

Tabla 6: Rendimiento FIR basado en bloques

En este caso la comparación de rendimientos viendo sólo las ecuaciones no es obvia. Vemos que en la versión original las dos variables (Filterlength y Blocklength) están divididas por dos, mientras que en el caso de la versión SIMD una está dividida por cuatro. Sin embargo, en la versión original, el bucle interno tiene dos instrucciones y en el caso de la versión SIMD, el bucle interno tiene tres. Para ver gráficamente la mejora que la nueva versión supone, miremos la siguiente ilustración:

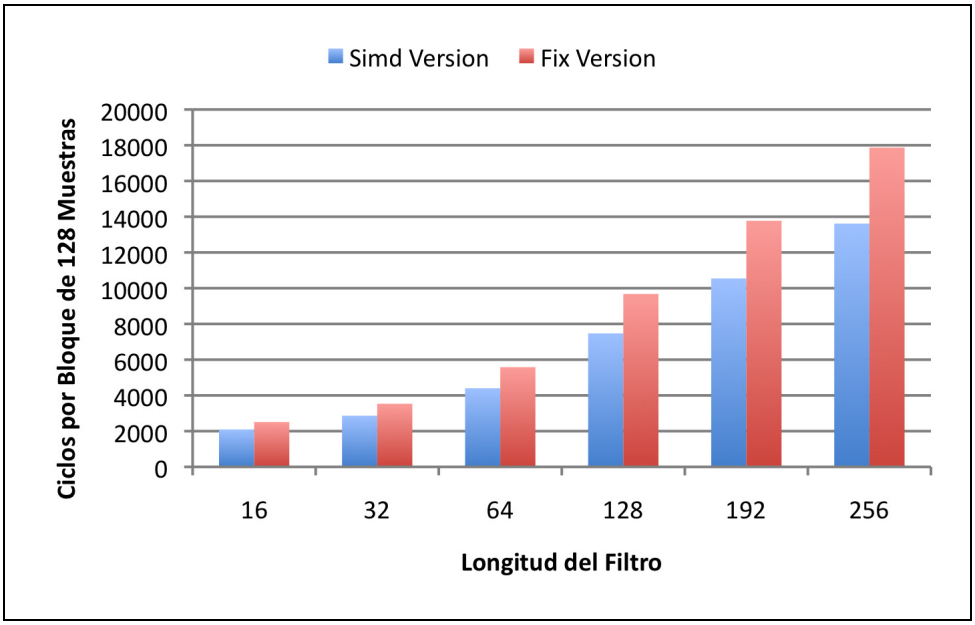

Ilustración 8: Comparación de rendimiento FIR basado en bloques.

4.1.3. FIR Basado en Bloques de Muestras Estéreo

El algoritmo utilizado originalmente en la librería matemática de *CoolFlux* para el filtro FIR basado en bloques de muestras estéreo aplica los mismos coeficientes a los dos canales, que están entrelazados en el bloque de entrada. Es necesario llevar dos vectores de retardo (uno por canal).

Como para las anteriores versiones, vamos a ver el funcionamiento de la función original:

Vector de coeficientes de tamaño 64 (64 palabras):

h_0	h_1	h_2	h_3	...	h_{63}

Dos vectores de retardos de tamaño 63 (63 palabras):

X_{n-1} (ch1)	X_{n-2} (ch1)	X_{n-3} (ch1)	...	X_{n-63} (ch1)

X_{n-1} (ch2)	X_{n-2} (ch2)	X_{n-3} (ch2)	...	X_{n-63} (ch2)

Bloque de entrada (256 palabras):

$X_{0\ (ch1)}$	$X_{0\ (ch2)}$	$X_{1\ (ch1)}$	$X_{1\ (ch2)}$	...	$X_{127(ch1)}$	$X_{127(ch2)}$

Cómputo de una muestra de salida:

$$Y_{n\ (ch1)} = X_{n\ (ch1)} * h_0 + X_{n-1\ (ch1)} * h_1 + \ldots + X_{n-63\ (ch1)} * h_{63}$$

$$Y_{n\ (ch2)} = X_{n\ (ch2)} * h_0 + X_{n-1\ (ch2)} * h_1 + \ldots + X_{n-63\ (ch2)} * h_{63}$$

Vectores de retardos después de computar el bloque:

$X_{127\ (ch1)}$	$X_{126\ (ch1)}$	$X_{125\ (ch1)}$	...	$X_{65\ (ch1)}$

$X_{127\ (ch2)}$	$X_{126\ (ch2)}$	$X_{125\ (ch2)}$	...	$X_{65\ (ch2)}$

Vector de salida:

$Y_{0\ (ch1)}$	$Y_{0\ (ch2)}$	$Y_{1\ (ch1)}$	$Y_{1\ (ch2)}$	...	$Y_{127(ch1)}$	$Y_{127(ch2)}$

Como en los anteriores casos para que el algoritmo funcione correctamente, los vectores de retardos se han de inicializar poniendo un cero en todas las posiciones. El algoritmo del FIR basado en bloques de muestras estéreo, es un algoritmo más lento que el basado es muestras mono. Es decir, procesar 128 muestras estéreo requiere un mayor número de ciclos que procesar 256 muestras mono. Esto es debido a que en la versión estéreo, es necesario llevar dos punteros que apuntan al vector de entras, una por canal, también hay que llevar dos vectores de retardos, lo que implica más tráfico con memoria y otros dos punteros para el vector de salida. En este caso es dónde el tipo *SIMD* destaca, puesto que no hace falta optimizar esta función que es más costosa que la versión mono. Lo que en este caso se va a hacer es adaptar la versión mono para que funcione en estéreo. La idea es muy sencilla, vamos a tomar la versión que trabaja con *fix* del filtro FIR basado en muestras de un canal, y vamos a cambiar el tipo de datos *fix* por *SIMD* sin alterar el algoritmo. Lo único necesario para que el algoritmo funcione correctamente es que en el vector de coeficientes, que son de tipo *SIMD*, los coeficientes estén duplicados en la parte alta y baja. Veamos cómo queda la función:

Vector de coeficientes de tamaño 64 (64 palabras):

h_0	h_1	h_2	h_3	...	h_{63}
h_0	h_1	h_2	h_3	...	h_{63}

Vector de retardos de tamaño 63 (63 palabras):

$X_{n-1(ch1)}$	$X_{n-2(ch1)}$	$X_{n-3(ch1)}$	...	$X_{n-63(ch1)}$
$X_{n-1(ch2)}$	$X_{n-2(ch2)}$	$X_{n-3(ch2)}$	...	$X_{n-63(ch2)}$

Bloque de entrada (128 palabras):

$X_{0\ (ch1)}$	$X_{1\ (ch1)}$	$X_{2\ (ch1)}$	$X_{3\ (ch1)}$	...	$X_{127(ch1)}$
$X_{0\ (ch2)}$	$X_{1\ (ch2)}$	$X_{2\ (ch2)}$	$X_{3\ (ch2)}$	...	$X_{127(ch2)}$

Cómputo de una muestra de salida:

$Y_{n(ch1)}$	=	$X_{n(ch1)}$	*	h_0	+	$X_{n-1(ch1)}$	*	h_1	+	...	+	$X_{n-63(ch1)}$	*	h_{63}
$Y_{n(ch2)}$	=	$X_{n(ch2)}$	*	h_0	+	$X_{n-1(ch2)}$	*	h_1	+	...	+	$X_{n-63(ch2)}$	*	h_{63}

Vector de retardos después de computar el bloque:

$X_{127(ch1)}$	$X_{126(ch1)}$	$X_{125(ch1)}$	...	$X_{65(ch1)}$
$X_{127(ch2)}$	$X_{126(ch2)}$	$X_{125(ch2)}$	...	$X_{65(ch2)}$

Vector de salida:

$Y_{0(ch1)}$	$Y_{1(ch1)}$	$Y_{2(ch1)}$	$Y_{3(ch1)}$	...	$Y_{127(ch1)}$
$Y_{0(ch2)}$	$Y_{1(ch2)}$	$Y_{2(ch2)}$	$Y_{3(ch2)}$	...	$Y_{127(ch2)}$

Ahora veamos las ecuaciones que indican los ciclos necesarios para aplicar el filtro a todas las muestras del bloque:

Nombre de la Función	Número de Ciclos
fir_2chil_bb_process (simd)	21 + Blocklength/2 * (28 + (Filterlength/2 – 2) * 2)
fir_2chil_bb_process (fix)	17 + Blocklength * (18 + (Filterlength – 2) * 2)

Tabla 7: Rendimiento FIR basado en bloques estéreo

Se puede apreciar, a primera vista, que la ganancia es realmente alta, de hecho es mayor a dos, esto se aprecia al ver que tanto Blocklength como Filterlength están divididos por dos en la nueva

función. Por ejemplo, para un filtro de 128 muestras por canal y 64 coeficientes, la función original necesitaría 18.193 ciclos mientras que la nueva, sólo necesitaría 5.653 ciclos, aunque una vez más la precisión de los resultados es la mitad.

La siguiente gráfica muestra el rendimiento de filtros de diferente longitud para un bloque de datos de 128 muestras, para las dos versiones:

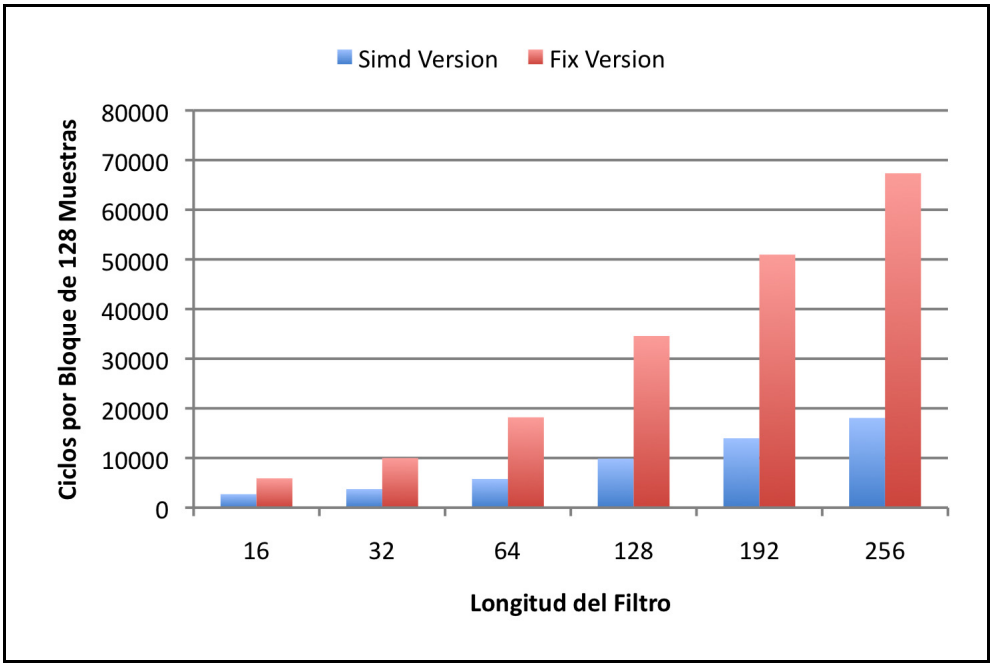

Ilustración 9: Comparación de rendimiento del FIR estéreo.

4.2. Filtro Biquad

El filtro biquad es un filtro lineal recursivo de segundo orden que contiene dos polos u dos ceros. "Biquad" es la abreviación de "biquadratic", que hace referencia al hecho que en el dominio Z, su función de trasferencia esta en la relación de dos funciones cuadráticas.

Los filtros recursivos de orden alto pueden ser muy sensibles a la cuantificación de sus coeficientes, y pueden volverse inestables fácilmente. Esto es mucho menos problemático en filtros de primer y segundo orden, por eso, los filtros de orden mayor suelen implementarse usando varios biquad en casacada y un filtro de primer orden si es necesario.

La implementación seguida es la forma directa 1 y sigue la siguiente ecuación:

$$y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) - a_1 y(n-1) - a_2 y(n-2)$$

Los coeficientes b_0, b_1 y b_2 determinan los ceros, y a_1, a_2 determinan la posición de los polos.

Se implementa la forma directa 1 porque da una mayor precisión y tiene menos riesgo de desbordamiento

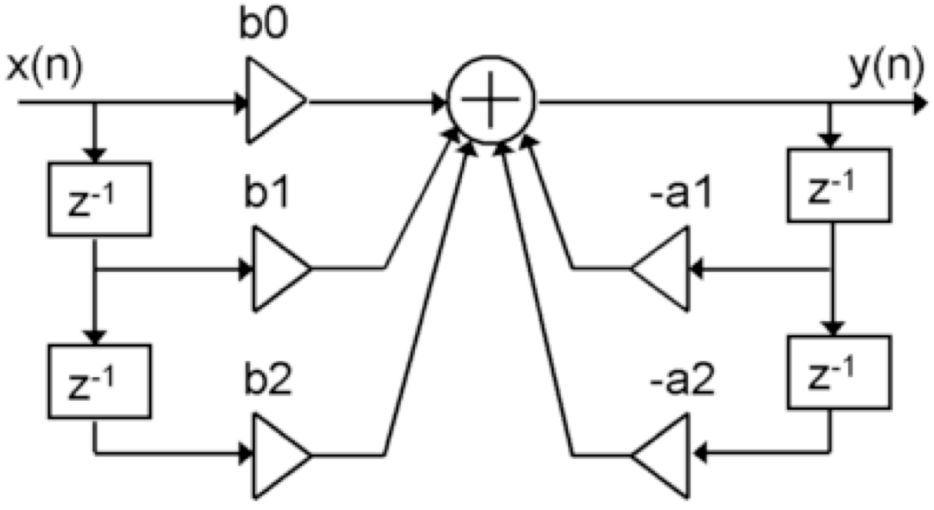

Ilustración 10: Filtro biquad en la forma directa 1

4.2.1. Biquad

En la versión original, la que trabaja con datos fix, hay implementadas tres funciones diferentes:

- Biquad_1ch_bb_1section_process: procesa una señal mono con un filtro biquad de una sola sección.
- Biquad_1ch_bb_nsection_process: procesa una señal mono con un filtro biquad de n secciones.
- Biquad_1ch_bb_step_scale_process: procesa un canal de datos con un filtro biquad de una sola sección. Puede ser utilizada para procesar señales de más de un canal. Esta función tiene como precondición que el número de muestras del bloque de entrada ha de ser par.

La función biquad_1ch_bb_1section_process recibe un bloque de datos de entrada, y un de salida, un entero que indica el número de elementos que tienen los bloques de datos recibe una lista con los cinco coeficientes del filtro y recibe tambien una lista de retardos (x(n-1) x(n-2) y(n-1) y(n-2)).

La función biquad_1ch_bb_nsection_process recibe los mismos parámetros que la anterior función añadiendo uno más, que es un entero que indica el número de secciones que tiene el filtro.

La versión biquad_1ch_bb_step_scale_process de la función recibe los mismos cinco primeros parámetros de las versiones anteriores, pero a esos se le añaden un stepIn que indica el desplazamiento necesario para conseguir el siguiente elemento del bloque de entrada, un stepOut que indica el desplazamiento en el bloque de salida y shift que es el desplazamiento que se ha aplicado a los coeficientes para escalarlos.

Viendo la ecuación del biquad se ve que es necesario utilizar la salida del filtro para procesar la siguiente muestra, esto es un problema a la hora de utilizar datos SIMD porque el hecho de tener dos datos en la misma muestra, hace que sea necesario separar ambos datos y volverlos a unir con otros, y esto nos hace perder mucho rendimiento.

Hay que tener en cuenta, que la función original, procesa dos muestras al mismo tiempo, lo que nos supone otro reto, porque para calcular las muestras siguientes necesitamos unir las salidas de las dos muestras que se procesan al mismo tiempo. Además al estar calculando dos muestras al mismo tiempo, el resultado de una de ellas se usara en el calculo de la otra.

El que la función original calcule dos muestras al mismo tiempo, hace necesario calcular cuatro en SIMD para poder obtener un mayor rendimiento en la función SIMD que en la original, que trabaja con datos fix.

También hay que tener en cuenta los retardos, que no son los mismos para las dos muestras que se calculan simultáneamente. Teniendo en cuenta todas las operaciones extras que hay que hacer para el cálculo de cuatro muestras simultáneamente con datos SIMD se llega a la conclusión que la versión resultante es bastante peor que la original que trabaja con datos fix.

Haciendo un cálculo rápido, el bucle principal de la función que trabaja con datos fix, tarda de 15 a 16 ciclos, es decir, nuestro objetivo para la función SIMD sería de unos 8 ciclos. Teniendo en cuenta que harían falta separar y unir datos para cada una de las muestras que se están calculando, es decir, 8 separaciones y 4 uniones. Esto son 12 operaciones extra respecto de la función original, que además no se pueden combinar con otras, esto hace que sean necesarios 12 ciclos más los 8 ciclos de nuestro valor ideal hacen un total de 20 ciclos, es decir nuestra nueva función sería bastante más lenta que la original.

Para poder sacar partido del uso de datos SIMD, en vez de procesar cuatro muestras mono, lo que se va a hacer es procesar dos muestras en estéreo, es decir dos muestras de un canal y otras dos muestras de otro. Al ser canales independientes no hay ningún problema en trabajar con datos SIMD ya que no va hacer falta separar los datos SIMD, el algoritmo es exactamente igual que el original.

Hacer que las funciones existentes trabajen en estéreo es muy sencillo, lo único que hay que hacer es cambiar en la función los tipos de datos de fix a SIMD y duplicar el vector de coeficientes, poniendo en la parte alta y en la parte baja, de cada dato SIMD del vector de coeficientes, el mismo valor. Con eso la función funciona correctamente para las tres versiones del filtro, conservando la precondición precondición de la funcion biquad_1ch_bb_step_scale_process.

Los nuevos nombres de las funciones serán los siguientes biquad_2ch_bb_1section_process, biquad_2ch_bb_nsection_process, biquad_2ch_bb_step_scale_process.

El siguiente problema en este módulo es validar las funciones. Como el filtro tiene realimentación, la pérdida de precisión producida por el uso de datos SIMD afecta a los resultados de las siguientes muestras, es decir, la diferencia entre los resultados entre los datos que produce la función que usa datos fix y los que produce la función que usa datos SIMD son cada vez mayores. Para solventar esto, hay que volver a

calcular nuevos datos de salida para la validación. Para volver a calcularlos lo que hay que hacer es truncar los datos de entrada para la función original, que ya está validada y sabemos que produce resultados correctos, y truncar también los valores a medida que se van calculando dentro de la función. Con los nuevos datos de salida la nueva función valida perfectamente.

La función biquad_2ch_bb_nsection_process, es la que más problemas da a la hora de validarla, dado que aquí los resultados de una sección se usan en la siguiente, es decir, la reutilización de datos es mayor, por tanto, el arrastre del error por falta de precisión más acusado.

Veamós ahora el rendimiento de cada una de las funciones, ahora comparar los rendimientos no es inmediato, puesto la nueva función trabaja con dos canales en vez de con uno sólo.

Nombre de la Función	**Número de Ciclos (bloqe par)**	**Número de Ciclos (bloqe impar)**
biquad_1ch_bb_1section_process	17 + (BlockSize/2) * 15	28 + (BlockSize/2) * 15
biquad_2ch_bb_1section_process	19 + (BlockSize/4)*17	30 + (BlockSize/4)*17
biquad_1ch_bb_nsection_process	14 + ((BlockSize/2)*15) * NumSections + (NumSections*12)	14 + ((BlockSize/2)*15) * NumSections + (NumSections*24)
biquad_2ch_bb_nsection_process	14 + ((BlockSize/4)*16) * NumSections + (NumSections*12)	14 + ((BlockSize/4)*16) * NumSections + (NumSections*25)
biquad_1ch_bb_step_scale_process	10 + (BlockSize/2)*16	No Permitido
biquad_2ch_bb_step_scale_process	12 + (BlockSize/4)*16	No Permitido

Tabla 8: Rendimiento filtro biquad

Los rendimientos de las funciones que trabajan con SIMD están puestos en función del número de muestras y no del tamaño del bloque de entrada, que es justo la mitad. Podemos ver cómo procesamos casi el doble de muestras, quizás se vea de una forma más clara en el siguiente gráfica. Calculada con bloques de 128 datos para los pares, 127 para los impares y para 3 secciones para biquad_xch_bb_nsection_process.

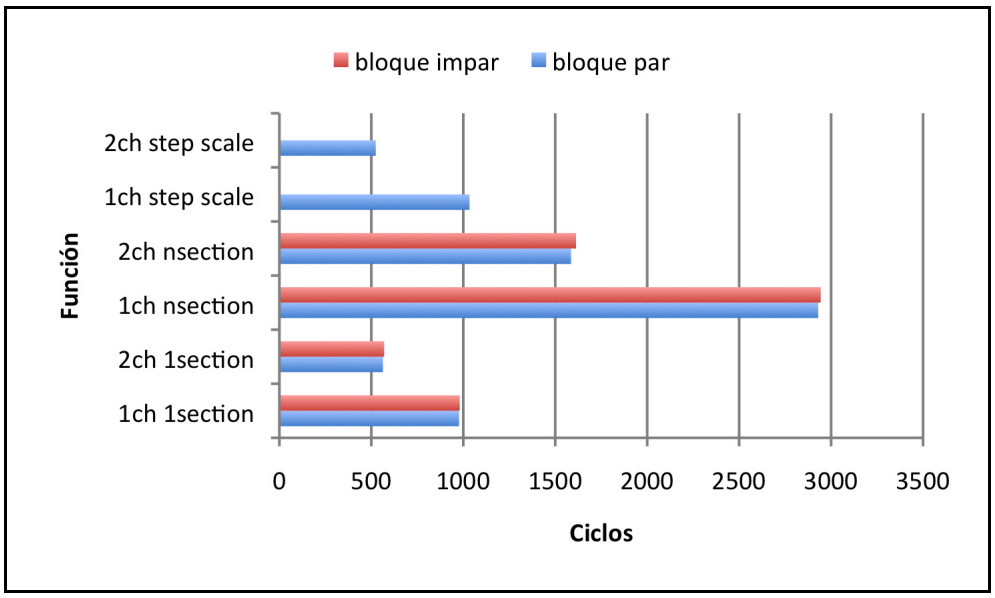

Ilustración 11: Comparación de rendimiento del biquad

4.2.2. Biquad DPFB

La función biquad DPFB o biquad double precision feedback, es la misma función que la anterior con la diferencia que la realimentación del filtro se hace con el doble de precisión, dada la importancia que tiene la exactitud de los datos de la realimentación. Esta función no pudo ser optimizada debido a un bug en la librería nativa. Dicho bug afectaba a la conversión de datos simd24, que producía dato erróneos. El hecho de utilizar datos de doble precisión para ésta versión del filtro, hace necesario utilizar dicha conversión.

La idea para optimizar esta función es la misma que la anterior, utilizar los datos SIMD para procesar muestras estéreo, ya que intentar procesar el doble de muestras en mono acabaría en una pérdida de rendimiento importante.

4.3. Operaciones con Vectores

La librería de *CoolFlux* incluye un conjunto de operaciones definidas para los vectores, originalmente escritas para vectores del tipo fix. El objetivo de esta parte del proyecto es adaptar las funciones existentes para que trabajen con vectores de datos SIMD y comparar el rendimiento de ambas versiones de cada operación. Además de las operaciones ya existentes, surgen nuevas operaciones con vectores de datos SIMD, que no tienen sentido para vectores de datos fix.

Las operaciones que son comunes, en ambos tipos de datos, son las siguientes:

- zerovector_Process: dado un vector de longitud N, la función inicializa con ceros todas sus posiciones.
- copyvector_Process: dados dos vectores de longitud N, la función copia en el segundo vector el contenido del primero.
- addvector_Process: dados tres vectores de longitud N, la función almacena en el tercero la suma de los dos primeros.
- subsector_Process: dados tres vectores de longitud N, la función almacena en el tercero la resta del segundo vector al primero de ellos.
- multvector_Process: dados tres vectores de longitud N, la función almacena en el tercero el resultado de multiplicar el primero por el segundo.
- addconstvector_Process: dados dos vectores de longitud N y una constante, la función almacena en el segundo el resultado de sumar la constante a todas las posiciones del primer vector.
- multconstvector_Process: dados dos vectores de longitud N y una constante, la función almacena en el segundo el resultado de multiplicar por la constante todas las posiciones del primer vector.
- shiftvector_Process: dados dos vectores de longitud N y un entero K, la función almacena en el segundo el resultado de hacer un desplazamiento de todos los elementos del primer vector K-veces.

Las nuevas operaciones introducidas para vectores de datos SIMD son las siguientes:

- joinvector_Process: dados dos vectores de tipo fix y longitud N, y un vector de tipo SIMD y longitud N, la función une los dos vectores de tipo fix en el vector de tipo SIMD.

- unjoinvector_Process: dado un vector de tipo SIMD y longitud N, y dos vectores de tipo fix y longitud N, la función divide el vector del tipo SIMD y copia la mitad de los datos en el primer vector de tipo fix, y la otra mitad en el segundo vector de tipo fix.
- dupvector_Process: dado un vector de tipo fix y longitud N, y un vector de tipo SIMD y longitud N, la función une el vector de tipo fix consigo misma y almacena el resultado en el vector de tipo SIMD.

4.3.1. zerovector_Process

Esta función en su versión original recibe dos parámetros, un vector del tipo fix y un entero N que indica la longitud del vector. Después de la ejecución, el vector contiene un cero en todas sus posiciones. Vamos a ver más detalladamente el funcionamiento del proceso.

Vector de tamaño N en la entrada:

¿?	¿?	¿?	¿?	…	¿?

Vector de tamaño N en la salida:

0	0	0	0	…	0

La función en su versión para SIMD recibe, también, dos parámetros de entrada, un vector del tipo SIMD y un entero N que nos indica la longitud del vector. Hay que tener en cuenta que los datos SIMD contienen dos valores, por consiguiente, en un vector de longitud N, tendremos 2N valores. Vamos a ver como el funcionamiento es muy similar al de la versión original:

Vector de tamaño N en la entrada:

¿?	¿?	¿?	¿?	…	¿?
¿?	¿?	¿?	¿?	…	¿?

Vector de tamaño N en la salida:

0	0	0	0	…	0
0	0	0	0	…	0

Vamos a ver los costes que esta operación tiene en sus dos versiones:

Nombre de la Función	Número de Ciclos
zerovector_Process (simd)	6 + N
zerovector_Process (fix)	6 + N

Tabla 9: Rendimiento zerovector_Process

Podemos ver que los costes en número de ciclos son iguales, pero si tenemos en cuenta que cada posición del vector contiene dos valores entonces el rendimiento de nuestra nueva función es prácticamente el doble.

4.3.2. copyvector_Process

Esta función recibe tres parámetros, dos vectores y un entero N que indica el tamaño de los dos vectores. Los dos vectores son del mismo tipo, fix en la versión original y SIMD en la nueva. La función lo que hace es una copia del primer vector en el segundo. Vamos a ver más detalladamente el funcionamiento del proceso con vectores del tipo fix.

Vectores de tamaño N en la entrada:

X_0	X_1	X_2	X_3	…	X_{N-1}

¿?	¿?	¿?	¿?	…	¿?

Vectores de tamaño N en la salida:

X_0	X_1	X_2	X_3	…	X_{N-1}

X_0	X_1	X_2	X_3	…	X_{N-1}

El funcionamiento con SIMD es exactamente el mismo, la única diferencia es que en un vector de longitud N tenemos almacenados 2N valores. La forma exacta de funcionar sería la que sigue.

Vectores de tamaño N en la entrada:

X_0	X_2	X_4	X_6	…	X_{2N-2}
X_1	X_3	X_5	X_7	…	X_{2N-1}

¿?	¿?	¿?	¿?	…	¿?
¿?	¿?	¿?	¿?	…	¿?

Vectores de tamaño N en la salida:

X_0	X_2	X_4	X_6	…	X_{2N-2}
X_1	X_3	X_5	X_7	…	X_{2N-1}

X_0	X_2	X_4	X_6	…	X_{2N-2}
X_1	X_3	X_5	X_7	…	X_{2N-1}

El rendimiento, evidentemente, va a ser el doble si tenemos en cuenta el número de valores almacenados en el vector y no el número de elementos del tipo base que contienen. Vamos a ver las ecuaciones que determinan el número de ciclos que tarda con vector de tamaño N en copiarse:

Nombre de la Función	Número de Ciclos
copyvector_Process (simd)	6 + N * 2
copyvector_Process (fix)	6 + N * 2

Tabla 10: Rendimiento copyvector_Process

4.3.3. addvector_Process, subsector_Process y multvector_Process

Estas funciones reciben cuatro parámetros, tres vectores del mismo tipo y un entero N que indica el tamaño de los tres vectores. Las funciones lo que hacen es coger un dato de la posición i-ésima del primer vector y un dato del segundo de la misma posición y almacenar el resultado de aplicar a esos dos valores una operación. Dicha operación puede ser una suma en el caso de addvector, una resta en el caso de subsector o una multiplicación en el caso de multvector. El resultado se almacenará en el tercero de los vectores, en su posición correspondiente. Veamos de una forma más clara, la forma en la que se realizan dichas operaciones, primero para el caso de vectores de datos fix, y después cómo se hace para el caso con datos SIMD. Podremos observar que el funcionamiento es el mismo, con la salvedad del número de valores que tenemos en los vectores de tamaño N, que en el caso de datos SIMD tendremos 2N valores, mientras que con fix sólo tenemos la mitad. Además del número de valores, también el número de operaciones, que en el mismo tiempo haremos el doble si estamos trabajando con datos SIMD.

Caso para datos fix:

Vectores de tamaño N en la entrada:

X_0	X_1	X_2	X_3	…	X_{N-1}
Y_0	Y_1	Y_2	Y_3	…	Y_{N-1}
¿?	¿?	¿?	¿?	…	¿?

Vectores de tamaño N en la salida:

X_0	X_1	X_2	X_3	…	X_{N-1}
Y_0	Y_1	Y_2	Y_3	…	Y_{N-1}
Y_0 OP X_0	Y_1 OP X_1	Y_1 OP X_1	Y_1 OP X_1	…	Y_{N-1} OP X_{N-1}

Caso para datos SIMD:

Vectores de tamaño N en la entrada:

X_0	X_2	X_4	X_6	…	X_{2N-2}
X_1	X_3	X_5	X_7	…	X_{2N-1}
Y_0	Y_2	Y_4	Y_6	…	Y_{2N-2}
Y_1	Y_3	Y_5	Y_7	…	Y_{2N-1}
¿?	¿?	¿?	¿?	…	¿?
¿?	¿?	¿?	¿?	…	¿?

Vectores de tamaño N en la salida:

X_0	X_2	X_4	X_6	…	X_{2N-2}
X_1	X_3	X_5	X_7	…	X_{2N-1}
Y_0	Y_2	Y_4	Y_6	…	Y_{2N-2}
Y_1	Y_3	Y_5	Y_7	…	Y_{2N-1}
Y_0 OP X_0	Y_2 OP X_2	Y_4 OP X_4	Y_5 OP X_5	…	Y_{2N-2} OP X_{2N-2}
Y_1 OP X_1	Y_3 OP X_3	Y_5 OP X_5	Y_7 OP X_7	…	Y_{2N-1} OP X_{2N-1}

Para obtener el máximo rendimiento en estas operaciones, es necesario que uno de los vectores de origen se encuentre almacenado en la XMEM mientras que el otro debe estar en YMEM. Al tener los vectores de origen en diferentes memorias, podemos leer de ambos vectores al mismo tiempo y en un solo ciclo y en el siguiente se hará la suma y la escritura en el destino. Estas son las ecuaciones que indican el número de ciclos necesarios para completar cada una de las funciones vistas en esta sección:

Nombre de la Función	Número de Ciclos
addvector_Process (simd)	6 + N * 2
addvector_Process (fix)	6 + N * 2
subvector_Process (simd)	6 + N * 2
subvector_Process (fix)	6 + N * 2
multvector_Process (simd)	7 + N * 2
multvector_Process (fix)	6 + N * 2

Tabla 11: Rendimiento addvector_Process, subsector_Process, multvector_Process

Vemos que en este caso el coste en ciclos es el aproximadamente el mismo, pero el rendimiento es el doble en el caso de vectores SIMD, puesto que contienen el doble de valores de los que contienen los vectores del tipo fix.

4.3.4. addconstvector_Process y multconstvector_Process

Las funciones addconstvector y multconstvector, reciben dos vectores del mismo tipo, uno de ellos será de entrada y el otro será de salida, también recibe un entero N que indica el tamaño de los vectores y recibe una constante del mismo tipo que el de los vectores. Lo que hacen estas funciones es sumar, en el caso de addconstvector, o multiplicar, en el caso de multconstvector, la constante a cada una de las posiciones del primer vector, y almacenar el resultado en el segundo vector. La manera de funcionar con el tipo fix y SIMD es la misma, pero el resultado puede no serlo. Es importante darse cuenta que los datos SIMD contienen dos valores, es decir, si la constante que entra en la función tiene valores diferentes en la parte alta y baja, las funciones no son equivalentes, porque estaríamos aplicando un valor a la mitad de los valores del vector, y otro valor diferenta a la otra mitad.

Si queremos que las funciones sean equivalentes, hay que asegurarse, antes de llamar a la función, que los valores de ambas

partes de la constante sean iguales. Es posible que en alguna situación busquemos el comportamiento diferente, ya que sería como manejar dos vectores y dos constantes al mismo tiempo.

Veamos el funcionamiento de las dos versiones de las funciones addconstvector y multconstvector.

Caso para datos fix:

Constante de entrada:

C

Vectores de tamaño N en la entrada:

X_0	X_1	X_2	X_3	…	X_{N-1}

¿?	¿?	¿?	¿?	…	¿?

Vectores de tamaño *N* en la salida:

X_0	X_1	X_2	X_3	…	X_{N-1}

X_0 OP C	X_1 OP C	X_2 OP C	X_3 OP C	…	X_{N-1} OP C

Caso para datos SIMD:

C_1
C_2

Vectores de tamaño *N* en la entrada:

X_0	X_2	X_4	X_6	…	X_{2N-2}
X_1	X_3	X_5	X_7	…	X_{2N-1}

¿?	¿?	¿?	¿?	…	¿?
¿?	¿?	¿?	¿?	…	¿?

Vectores de tamaño *N* en la salida:

X_0	X_2	X_4	X_6	…	X_{2N-2}
X_1	X_3	X_5	X_7	…	X_{2N-1}

X_0 OP C_1	X_2 OP C_1	X_4 OP C_1	X_5 OP C_1	…	X_{2N-2} OP C_1
X_1 OP C_2	X_3 OP C_2	X_5 OP C_2	X_7 OP C_2	…	X_{2N-1} OP C_2

Las siguientes ecuaciones determinan el número de ciclos necesarios para efectuar las operaciones descritas en esta sección.

Nombre de la Función	Número de Ciclos
addconstvector_Process (simd)	7 + N * 2
addconstvector_Process (fix)	6 + N * 2
multconstvector_Process (simd)	7 + N * 2
multconstvector_Process (fix)	6 + N * 2

Tabla 12: Rendimiento addconstvector_Process, multconstvector_Process

Al igual que en las operaciones anteriores, podemos decir que el rendimiento es el doble ya que los vectores del tipo SIMD contienen el doble de elementos que los de tipo fix.

4.3.5. shiftvector_Process

Esta función recibe cuatro parámetros, dos vectores del mismo tipo un entero N que indica la longitud de los vectores y el cuarto parámetro es otro entero K que indica el número de posiciones que hay que desplazar cada valor. Lo que hace la función shiftvector es coger un valor del primer vector y desplazarlo K posiciones hacia la izquierda (<<), si K es positivo, si no el desplazamiento será hacia la derecha (>>). El resultado del desplazamiento se hará pasar a través de la unidad RSS porque es posible que sea necesario redondearlo.Esta función es igual para las versiones fix y SIMD, por supuesto que la versión que maneja datos SIMD opera con el doble de valores.

Versión para datos fix:

Vectores de tamaño N en la entrada:

X_0	X_1	X_2	X_3	…	X_{N-1}

¿?	¿?	¿?	¿?	…	¿?

Vectores de tamaño N en la salida:

X_0	X_1	X_2	X_3	…	X_{N-1}

$X_0<<K$	$X_1<<K$	$X_2<<K$	$X_3<<K$	…	$X_{N-1}<<K$

Versión para datos SIMD:

Vectores de tamaño N en la entrada:

X_0	X_2	X_4	X_6	…	X_{2N-2}
X_1	X_3	X_5	X_7	…	X_{2N-1}

¿?	¿?	¿?	¿?	…	¿?
¿?	¿?	¿?	¿?	…	¿?

Vectores de tamaño N en la salida:

X_0	X_2	X_4	X_6	…	X_{2N-2}
X_1	X_3	X_5	X_7	…	X_{2N-1}

X_0<<K	X_2<<K	X_4<<K	X_6<<K	…	X_{2N-2}<<K
X_1<<K	X_3<<K	X_5<<K	X_7<<K	…	X_{2N-1}<<K

Las siguientes ecuaciones determinan el número de ciclos necesarios para efectuar las operaciones descritas en esta sección.

Nombre de la Función	Número de Ciclos
shiftvector_Process (simd)	10 + N * 2
shiftvector_Process (fix)	10 + N * 2

Tabla 13: Rendimiento shiftvector_Process

Al igual que en las operaciones anteriores, podemos decir que el rendimiento es el doble ya que los vectores del tipo SIMD contienen el doble de elementos que los de tipo fix.

4.3.6. joinvector_process

Esta función sólo está disponible para vectores del tipo SIMD. Recibe cuatro parámetros: dos vectores del tipo fix, que van a ser los vectores origen, un vector del tipo SIMD, que será el vector destino, y un entero N que indica la longitud de los vectores. La función tomará los valores de la posición i-ésima de cada vector de entrada, los unirá en un dato tipo SIMD, mediante el uso de la instrucción join_simd12, y el resultado lo almacenará en la posición i-ésima del vector de destino. Para poder hacer esto de la manera más eficiente posible, uno de los

vectores de origen debe estar en la XMEM, mientras que el otro vector de origen debe estar en la YMEM, esto nos permite hacer dos lecturas simultáneamente. Veamos qué hace la operación con los parámetros que recibe.

Vectores de tamaño N en la entrada:

X_0	X_1	X_2	X_3	…	X_{N-1}

Y_0	Y_1	Y_2	Y_3	…	Y_{N-1}

¿?	¿?	¿?	¿?	…	¿?
¿?	¿?	¿?	¿?	…	¿?

Vectores de tamaño N en la salida:

X_0	X_1	X_2	X_3	…	X_{N-1}

Y_0	Y_1	Y_2	Y_3	…	Y_{N-1}

X_0	X_1	X_2	X_3	…	X_{N-1}
Y_0	Y_1	Y_2	Y_3	…	Y_{N-1}

Los ciclos necesarios para la generación de cada posición del vector destino son tres, uno para realizar la lectura de los vectores de origen, el segundo ciclo, para unir los dos datos leídos, y el tercero, guarda en el vector destino el dato generado por la unión anterior. Veamos cuál es el número de ciclos necesarios para aplicar la operación sobre vectores de longitud N:

Nombre de la Función	Número de Ciclos
joinvector_Process	8 + N * 3

Tabla 14: Rendimiento joinvector_Process

4.3.7. unjoinvector_Process

Esta operación es la opuesta a joinvector, ésta recibe como parámetro tres vectores, como en el caso anterior, pero aquí el vector de origen es el de tipo SIMD y los de destino son los de tipo fix, y también recibe un entero N que indica la longitud de los vectores. Para su funcionamiento, uno de los vectores de destino debe estar en la XMEM

y el otro debe estar en la YMEM. Que los vectores de destino estén en memorias diferentes nos permite escribir en los dos vectores destino en un solo ciclo. Lo que hace la función, es separar la parte alta y la baja del dato SIMD del vector origen y almacena cada una de las partes en uno de los vectores destino.

Vectores de tamaño N en la entrada:

X_0	X_1	X_2	X_3	…	X_{N-1}
Y_0	Y_1	Y_2	Y_3	…	Y_{N-1}

¿?	¿?	¿?	¿?	…	¿?

¿?	¿?	¿?	¿?	…	¿?

Vectores de tamaño N en la salida:

X_0	X_1	X_2	X_3	…	X_{N-1}
Y_0	Y_1	Y_2	Y_3	…	Y_{N-1}

X_0	X_1	X_2	X_3	…	X_{N-1}

Y_0	Y_1	Y_2	Y_3	…	Y_{N-1}

El coste en ciclos de esta función es mayor que la de joinvector, esto es debido a que en vez de realizar una operación de juntar los datos, hay que hacer dos de separar, dado que no existe una instrucción en *CoolFlux Complex* que nos permita obtener ambos datos en una sola instrucción. Además, las instrucciones de separar y juntar no pueden ser combinadas con otras para poder hacerlas en un solo ciclo. El bucle principal, entonces, tardará cuatro ciclos, en el primero se lee un dato del vector origen, en el segundo se extrae la parte alta del dato leído, en el tercero se extrae la parte baja, y en el cuarto se escriben los dos vectores destino, en uno la parte alta y en el otro la baja. La siguiente ecuación pone de manifiesto el número de ciclos que se necesitan en función de la longitud de los vectores.

Nombre de la Función	**Número de Ciclos**
unjoinvector_Process	8 + N * 4

Tabla 15: Rendimiento joinvector_Process

4.3.8. dupvector_Process

Esta operación, en esencia, es igual que joinvector dado que el objetivo es el de unir dos vectores en uno. La diferencia radica en que los vectores que se quiere unir son iguales. Esto es útil, por ejemplo, para duplicar los coeficientes del filtro FIR estéreo. La función, esta vez, recibe tres parámetros. Recibe dos vectores, uno del tipo fix, que es el que va a ser unido consigo mismo; el segundo vector es el vector destino, que es del tipo SIMD y por último, recibe un entero N que indica la longitud de los vectores. Vamos a ver que, efectivamente, lo que hace ésta operación es muy similar a joinvector.

Vectores de tamaño N en la entrada:

X_0	X_1	X_2	X_3	…	X_{N-1}

¿?	¿?	¿?	¿?	…	¿?
¿?	¿?	¿?	¿?	…	¿?

Vectores de tamaño N en la salida:

X_0	X_1	X_2	X_3	…	X_{N-1}

X_0	X_1	X_2	X_3	…	X_{N-1}
X_0	X_1	X_2	X_3	…	X_{N-1}

En éste caso, el coste en ciclos de la operación no está sujeto a la ubicación de los vectores en memoria, puesto que leemos de un solo vector y escribimos en un solo vector. El bucle principal requiere tres ciclos, uno para la lectura, otro para la unión y el último para la escritura. La siguiente ecuación indica el coste en ciclos de aplicar esta operación a vectores de tamaño N.

Nombre de la Función	Número de Ciclos
dupvector_Process	8 + N * 3

Tabla 16: Rendimiento dupvector_Process

4.4. IDCT

La transformada discreta del coseno inversa (IDCT) es la función inversa de la transformada discreta del coseno, que es una transformada basada en la transformada de Fourier discreta (DFT), pero utilizando únicamente números reales.

Existen diferentes variantes de la DCT, pero la versión implementada en el proyecto es la inversa de la DCT-II. Esta variante es la más común y normalmente se hace referencia a ella como DCT. La inversa de la DCT, que es la que vamos a implementar, corresponde a la DCT-III y es la que recibe el nombre de IDCT.

La expresión matemática de la IDCT:

$$X_k = \frac{1}{2} x_0 + \sum_{n=1}^{N-1} x_n \cos\left[\frac{\pi}{N} n\left(k + \frac{1}{2}\right)\right] \qquad k = 0,\ldots,N-1$$

Este tipo de transformaciones se suelen aplicar sobre matrices de 8x8 elementos, pero como podemos ver, la función matemática es lineal. Para aplicar la transformada sobre una matriz, primero se aplica por filas y el resultado de aplicarlo por filas es otra matriz de 8x8 a la cual le volvemos a aplicar la misma función, pero esta vez por columnas y el resultado de la aplicación de la función por columnas, es el resultado que buscábamos. Esto es posible gracias a que, al igual que la FFT, la DCT es una transformada separable.

Una se las aplicaciones más frecuentes de la DCT es en la compresión de imágenes, ya que la mayoría de los valores devueltos por la DCT son muy próximos a cero, y esto permite un elevado grado de compresión. Las transformaciones DCT son transformaciones sin pérdida, es decir, aplicando la inversa de la transformación obtienes los datos de partida exactos. La pérdida en algoritmos de compresión basados en la DCT, como JPEG, se producen en otra fase de la compresión que es la cuantización.

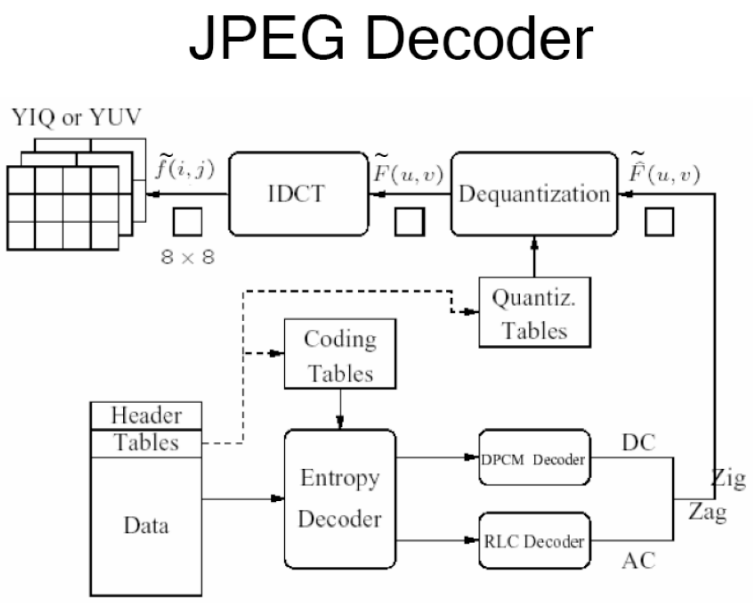

Ilustración 12: Intervención de la IDCT en la decodificación JPEG

4.4.1. Desarrollo

Está función no estaba implementada previamente en la librería matemática de *CoolFlux Complex*, así que lo primero que hubo que hacer fue implementar la función iDCT utilizando datos del tipo Fix, para después poder evaluar la mejora de rendimiento que suponía utilizar tipos de datos SIMD. Se realizaron dos versiones de la función, ambas utilizando la implementación fast iDCT, que es más eficiente computacionalmente debido al número de operaciones de sumas y multiplicaciones necesarias. Una de las implementaciones se hizo con una precisión de 24 bits, la otra de 12 bits. El motivo de realizar la versión de 12 bits era el de observar cual era exactamente el beneficio de utilizar el tipo de datos SIMD con los cuales se obtienen resultados de una precisión de 12 bits, dicho de otra manera trabajar en las mismas condiciones.

La función implementada utilizando el tipo de datos *fix*, recibe como parámetros tres vectores de 64 (8x8) posiciones. El primer vector, será el que contenga los datos de entrada, que serán los coeficientes de una DCT. El segundo vector, será un vector que se utilice para almacenar datos temporales, que serán los resultados de aplicar la IDCT por filas a los datos de entrada, éste vector deba estar alineado en memoria en un múltiplo de una potencia de dos, la cual ha de ser mayor o igual que el número de elementos del vector. El tercero, será la matriz de 8x8 original, antes de aplicar la DCT, obtenida aplicando por columnas la IDCT al vector de datos temporales.

La aplicación por filas de la función sobre la matriz es sencilla, suponiendo que el vector que representa la matriz, almacena dicha matriz en filas unas seguidas de otras, es decir, las posiciones de 0 a 7 serían de la primera fila, de 8 a 15 la segunda y así con las ocho filas de ocho elementos. Para ésta parte, el acceso al vector es completamente lineal y no presenta problemas a la hora de obtener un buen rendimiento.

La segunda aplicación de la IDCT, la que se hace por columnas es más problemática debido al acceso no secuencial de la memoria. Por ejemplo, para acceder a la primera columna, tenemos que acceder a las posiciones 0, 8, 16, 24, 32, 40, 48 y 56 del vector. Esto presenta un problema, porque hay que calcular las posiciones, y esto supone una carga operacional de la función que no corresponde al propio algoritmo de la IDCT. Utilizando el direccionamiento bit-reverse que nos ofrece *CoolFlux Complex*, el acceso a estos datos no supone mayor carga computacional que el acceso secuencial. Para poder hacer uso de esta función, el vector debe estar alineado en memoria como se ha descrito anteriormente.

Para realizar el direccionamiento *bit-reverse*, llamaremos a la función *reverse_incr* que recibe como parámetros el puntero a actualizar, un puntero que apunta al comienzo del vector (base) y un entero que indica el tamaño del vector y devuelve el nuevo valor del puntero. La llamada a esta función, con un vector de tamaño 64, ocho veces consecutivas, nos devolverá las siguientes posiciones del vector: base + 0, base + 32, base + 16, base + 24, base + 8, base + 40, base + 48, base + 56. Como podemos ver nos da los datos de un columna, aunque nos los da en desorden, algo poco importante comparado con el beneficio en rendimiento que esto supone.

Una vez terminada esta implementación y probada su corrección, lo siguiente es pensar el la implementación para *SIMD*. Que la IDCT sea un algoritmo separable hace que los datos *SIMD* puedan utilizarse de una forma muy eficiente.

La idea básica para sacar partido a los datos *SIMD*, es unir dos filas fix en una sola. Aplicando la transformada a una fila de *SIMD* lo estaremos haciendo sobre el equivalente a dos, esto es posible porque todas las operaciones necesarias para el calculo de la IDCT, son sumas, multiplicaciones y desplazamientos; operaciones soportadas por los tipos de datos *SIMD*.

Por el momento, parece que la mejora en rendimiento va a ser muy alta, cercana al doble, pero aun no hemos analizado los problemas que se presentan. El único problema que se nos presenta a la hora de la

implementación, es el acceso por columnas a la matriz para poder aplicar la IDCT.

Al haber agrupado los datos por filas, hace que al acceder por columnas no podamos obtener dos columnas al mismo tiempo, si no que obtenemos una columna en cuatro posiciones. Con esta configuración, no se puede aplicar la IDCT.

X_{00}	X_{01}	X_{02}	X_{03}	X_{04}	X_{05}	X_{06}	X_{07}
X_{10}	X_{11}	X_{12}	X_{13}	X_{14}	X_{15}	X_{16}	X_{17}
X_{20}	X_{21}	X_{22}	X_{23}	X_{24}	X_{25}	X_{26}	X_{27}
X_{30}	X_{31}	X_{32}	X_{33}	X_{34}	X_{35}	X_{36}	X_{37}
X_{40}	X_{41}	X_{42}	X_{43}	X_{44}	X_{45}	X_{46}	X_{47}
X_{50}	X_{51}	X_{52}	X_{53}	X_{54}	X_{55}	X_{56}	X_{57}
X_{60}	X_{61}	X_{62}	X_{63}	X_{64}	X_{65}	X_{66}	X_{67}
X_{70}	X_{71}	X_{72}	X_{73}	X_{74}	X_{75}	X_{76}	X_{77}

Ilustración 13: Matriz 8x8 representada con datos SIMD (por filas)

La situación de la ilustración no nos permite aplicar por columnas la transformada. La situación que buscamos sería la que se muestra a continuación.

X_{00}	X_{02}	X_{04}	X_{06}
X_{01}	X_{03}	X_{05}	X_{07}
X_{10}	X_{12}	X_{14}	X_{16}
X_{11}	X_{13}	X_{15}	X_{17}
X_{20}	X_{22}	X_{24}	X_{26}
X_{21}	X_{23}	X_{25}	X_{27}
X_{30}	X_{32}	X_{34}	X_{36}
X_{31}	X_{33}	X_{35}	X_{37}
X_{40}	X_{42}	X_{44}	X_{46}
X_{41}	X_{43}	X_{45}	X_{47}
X_{50}	X_{52}	X_{54}	X_{56}
X_{51}	X_{53}	X_{55}	X_{57}
X_{60}	X_{62}	X_{64}	X_{66}
X_{61}	X_{63}	X_{65}	X_{67}
X_{70}	X_{72}	X_{74}	X_{76}
X_{71}	X_{73}	X_{75}	X_{77}

Ilustración 14: Matriz 8x8 representada con datos SIMD (por columnas)

Con la matriz organizada por columnas, podemos procesar dos columnas, leyendo una de *SIMD*. El problema es cómo llegar de la organización de los *SIMD* por filas a la organización por columnas.

La solución a cómo llegar a la segunda matriz es, una vez leída y procesada una fila de la matriz de entrada, cogemos dos valores que vayan consecutivos en la fila, es decir los elementos *i* e *i + 1*, se paramos sus partes altas, las unimos y lo guardamos en la posición i, luego hacemos los mismo para las partes bajas y las guardamos en la posición *i + 4*. Es importante tener preste que aunque se hable de matrices, estamos trabajando con un vector en todo momento.

Una vez que tengamos el vector organizado como en la ilustración anterior, es decir, organizado por columnas, podemos acceder por columnas utilizaremos de nuevo la función *reverse_incr*, teniendo que implementar las restricciones que su uso impone al vector intermedio, la única diferencia es que ahora el parámetro que indica el tamaño en vez de ser 64 será 32 puesto que nuestro vector de *SIMD* es la mitad, aunque contenga el mismo número de datos.

El haber cambiado organización de la matriz para el proceso por columnas, puesto que en la salida queremos el mismo formato que en la entrada. Esto nos obliga a tener que invertir la operación que hemos tenido que hacer para volver a tener la matriz organizada por filas.

Para deshacer el cambio se hace exactamente lo mismo, la diferencia es que ahora los elementos consecutivos no son respecto a una fila sino que a una columna y al guardarlo se hace en posiciones consecutivas respecto a las filas, es decir, cogemos los elementos *i* e *i + 4*, separamos sus partes altas las unimos y guardamos el resultado en *i*, y las partes bajas las guardamos en *i + 1*. Con esto ya tenemos el resultado de aplicar la IDCT y en el mismo formato que la entrada.

Ahora ya tenemos una función que calcula una IDCT utilizando el tipo de datos *SIMD*, ahora veamos que ganancia en rendimiento nos supone. La parte que se encarga de hacer los cálculos, es decir, las sumas, multiplicaciones y desplazamientos, tarda aproximadamente la mitad. Lo que supone un problema de rendimiento es la reorganización de los datos necesaria para poder procesar las columnas en la versión *SIMD*, que no es necesaria en la *fix*.

Es en la reorganización donde perdemos prácticamente toda la ganancia que hemos obtenido. Esto es debido a que la separación y unión de datos *SIMD* en *CoolFlux Complex*, es muy ineficiente, entre otras cosas porque este tipo de instrucciones no pueden ejecutarse al mismo tiempo que otras instrucciones como pasa con otras operaciones que pueden combinarse con actualizaciones de punteros u otras

operaciones que se ejecuten por una ruta diferente. Además a causa de la reorganización, el patrón de accesos a memoria para guardar los resultados, tanto intermedio como final, es diferente y menos eficiente ya que el nuevo no es secuencial como en el caso de la versión *fix*. Por todo lo anterior, tenemos que la versión que trabaja con *SIMD* de la IDCT tenga prácticamente el mismo rendimiento que la que trabaja con *fix*.

Nombre de la Función	**Número de Ciclos**
idct_ifast (fix)	1244
Idct_ifast (simd)	1200

Tabla 17: Rendimiento de la IDCT

5. Conclusiones

Como se ha visto en los diferentes módulos en los que se ha trabajado, el rendimiento en general, obtenido mediante la introducción de datos *SIMD* en las nuevas funciones, es bueno. Lo bueno que sea el rendimiento, depende mucho del algoritmo que estemos aplicando. En el caso del filtro FIR la mejora del rendimiento ha sido bueno para las versiones mono, porque se ha conseguido adaptar el algoritmo de tal forma que no es necesario hacer uso de las operaciones de extracción o de unión. Para la versión en estéreo el rendimiento ha sido mejor de lo esperado, puesto que funcionando con el algoritmo para muestras en mono, se ha podido calcular muestras en estéreo. En el filtro biquad, no se ha podido encontrar una adaptación del algoritmo capaz de funcionar con muestras mono obteniendo un mejor rendimiento que en su versión original. Aquí el rendimiento se ha ganado adaptando la función para que trabaje con dos canales simultáneamente. En las operaciones con vectores se obtiene un buen rendimiento, ya que este tipo de operaciones se aplica sobre cada uno de los valores contenidos en el vector sin tener en cuenta el resto de los valores, que es cuando los datos *SIMD* ofrecen una gran ventaja, puesto que la aplicación de una operación sobre un dato *SIMD* tiene efecto sobre dos valores al mismo tiempo. En la IDCT no se ha podido ganar nada sobre la función que trabaja sobre datos del tipo *fix*. En principio el rendimiento que se podía ganar en esta función era bueno, ya que se puede aplicar la función sobre vectores independientes, un escenario muy bueno para el tipo de datos *SIMD*, pero luego llega a parte de la función en la que hay que aplicar la ecuación de la IDCT en vez de por filas, por columnas. Esto requiere una reestructuración de la matriz que necesita que se separen todos los datos *SIMD* y se junten de una manera diferente. Debido a la reestructuración de la matriz, la cual hay que realizar dos veces, se pierde prácticamente todo el rendimiento que se supuestamente se podía ganar.

De lo anteriormente explicado, sacamos como conclusión que los datos *SIMD* dan un buen rendimiento siempre y cuando no haya que cambiar el orden de los datos, es decir, cuando no haya que extraer partes y volverlas a juntar. Es decir, si queremos sacar un mayor rendimiento a las funciones que en este trabajo no han obtenido buenos resultados tras su adaptación a *SIMD*, es necesario mejorar el aspecto de la reordenación de datos tan ineficiente que *CoolFlux Complex* tiene

para el tipo SIMD. Por ejemplo, una posible mejora es que las operaciones de extraer y juntar, se puedan ejecutar en combinación con otras operaciones mientras la ruta de datos lo permita como con una actualización de índices. Otra posible mejora que tendría un impacto muy positivo en la IDCT es una nueva instrucción del procesador en la que dados dos datos *SIMD* el resultado de su ejecución, fuese que uno de ellos contuviese las partes altas y el otro las partes bajas. En la ilustración siguiente, podemos ver gráficamente cómo funcionaría la nueva instrucción propuesta.

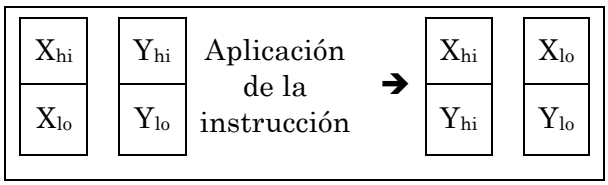

Ilustración 15: Esquema de la instrucción propuesta

Si esta instrucción estuviese disponible, tardaría un ciclo en ejecutarse siempre que los datos SIMD sobre los que se aplica la instrucción estén en bancos de registros diferentes ya que es preciso realizar dos escrituras. Con esta instrucción sustituiríamos cuatro instrucciones de extraer y dos de juntar, es decir, se ahorrarían cinco instrucciones (cinco ciclos) de cada seis. Si aplicamos esto a una matriz de 32 elementos SIMD, dos veces, como es necesario en la IDCT, en vez de tener un coste de tres instrucciones por elemento, se pasaría a media instrucción por elemento. Hablando en cifras totales, se pasaría de gastar 192 ciclos a sólo 32. Si además de la instrucción propuesta, se pudiera combinar con actualizaciones de punteros de las AGUs la ganancia sería mucho mayor.

Concluyendo, la incorporación de éste nuevo tipo de datos es muy positivo, ya que es posible obtener una gran mejora de rendimiento, en algunos casos, introduciendo pocas modificaciones sobre la arquitectura de *CoolFlux*. Por tanto, estas modificaciones son buenas candidatas a permanecer en la arquitectura de futuras revisiones de *CoolFlux*, auque es cierto que para que los datos SIMD puedan dar un mejor rendimiento en funciones cómo la IDCT, hay que introducir una serie de instrucciones que permitan un manejo mucho más ágil de los datos.

6. Bibliografía

[1] Feig, E., & Winograd, S. (1992). Fast Algorithms for the Discrete Cosine Transform. *IEEE Transactions on Signal Processing , 40* (9).

[2] NXP. (2007). *CoolFlux Complex DSP Assembly Programmer's Manual.*

[3] NXP. (2007). *CoolFlux Complex DSP: C Programmer's Manual* .

[4] Smith, S. W. (1997). *The Scientist and Engineer's Guide to Digital Signal Processing.*

[5] Texas Instruments. (Octubre de 2003). TMS320C64x DSP Library Programmer's Reference.

[6] Wikipedia. (2007). *Digital biquad filter*. Obtenido de Wikipedia: http://en.wikipedia.org/wiki/Digital_biquad_filter

[7] Wikipedia. (2007). *Discrete Cosine Transform*. Obtenido de Wikipedia: http://en.wikipedia.org/wiki/Discrete_cosine_transform

[8] Wikipedia. (2008). *Filtro Digital*. Obtenido de Wikipedia: http://es.wikipedia.org/wiki/Filtro_digital

[9] Wikipedia. (2007). *FIR*. Obtenido de Wikipedia: http://es.wikipedia.org/wiki/FIR

[10] Wikipedia. (2008). Procesador Digital de Señal. Obtenido de Wikipedia: http://es.wikipedia.org/wiki/Procesador_digital_de_señal